Analiza i modelowanie wydajności obliczeń

Część I. Obliczenia sekwencyjne (jednowątkowe)

Krzysztof Banaś

Spis treści

Rozdział 1

Wstęp

Wydajność jest jedną z najważniejszych własności oprogramowania (obok takich cech jak np. poprawność, niezawodność, bezpieczeństwo). Optymalizacja oprogramowania pod kątem wydajności (zwana w dalszej części książki po prostu optymalizacją) oznacza dążenie do skrócenia czasu osiągnięcia pożądanego wyniku (*time-to-solution*). Czas uzyskania wyniku, zwany dalej czasem wykonania, jest rozumiany jako czas przekształcenia danych wejściowych dostarczanych programowi w dane wyjściowe, żądane przez użytkownika.

Czas wykonania jest podstawowym parametrem analizowanym w badaniach nad wydajnością obliczeń. Ostatecznym celem modelowania wydajności jest uzyskanie wzorów pozwalających oszacować czas wykonania konkretnego programu dla konkretnych danych wejściowych. Istotną cechą uzyskiwanych oszacowań ma być uwzględnienie właściwości implementacji, dokonanej w konkretnym środowisku programowania, oraz charakterystyk sprzętu, na którym dokonywane są obliczenia. W tym miejscu modelowanie wydajności (*performance modelling*) różni się istotnie od klasycznej analizy złożoności obliczeniowej. W tej ostatniej chodzi o uzyskanie oszacowania czasu wykonania, w którym ważny jest rząd funkcji wyrażającej czas wykonania w zależności od rozmiaru danych wejściowych (ten ostatni rozumiany jest często w specyficzny sposób, odpowiedni dla analizowanych algorytmów). W takim ujęciu uwzględnienie implementacji i sprzętu mieści się w stałych, najczęściej niepodlegających szacowaniu, pojawiających się przy wyrażeniach zawierających parametr określający rozmiar danych.

W analizie wydajności dąży się do uzyskania wzorów wyrażających czas wykonania programów bez nieokreślonych stałych. Oznacza to próby ilościowego ujęcia wpływu własności sprzętu komputerowego wraz z jego oprogramowaniem, systemowym i narzędziowym, na wykonanie programów. Dokonywane jest to poprzez odpowiedni dobór specjalnych parametrów, uzyskiwanych teoretycznie lub eksperymentalnie, umieszczanych we wzorach na czas wykonania.

Wydajność przetwarzania, w najbardziej ogólnym ujęciu, można określić jako odwrotność czasu wykonania. Im krótszy czas wykonania programu, tym wyższa osiągnięta wydajność. Problemem w badaniu wydajności jest fakt, że czas wykonania jest wielkością łatwą do zdefiniowania i zmierzenia, podczas gdy wydajność zazwyczaj ujmuje już specyfikę dziedziny zastosowań i konkretnego badanego programu. W hipotetycznym przypadku programu realizującego sekwencję takich samych operacji, wydajność przetwarzania może być definiowana jako liczba operacji wykonywanych w jednostce czasu. Czas wykonania jest wtedy ilorazem liczby operacji w programie przez wydajność przetwarzania. Najczęściej w praktyce, uzyskanie oszacowań czasu wykonania wymaga zastosowania znacznie bardziej złożonych wzorów i miar wydajności.

Dokonywane w książce analizy wydajności dotyczą programów, a więc implementacji algorytmów w środowiskach programowania. Algorytm jest tutaj rozumiany jako ogólny przepis rozwiązania problemu obliczeniowego. Podstawowym, stosowanym w książce, sposobem zapisu algorytmów jest pseudokod, mający być intuicyjnie zrozumiały (do nadania struktury zapisowi algorytmów stosowane są standar-

dowe konstrukcje wykorzystywane w najpopularniejszych językach programowania). Drobne modyfikacje algorytmu dokonywane w różnych implementacjach, jak np. często stosowana zmiana kolejności wykonywania pętli, są traktowane jako nie zmieniające istoty algorytmu. W tym ujęciu złożoność obliczeniowa algorytmu pozostaje bez zmian, różnią się natomiast oszacowania czasu wykonania dla konkretnych implementacji (konkretnych programów).

W klasycznej analizie złożoności obliczeniowej, oszacowania wyrażane są najczęściej liczbą tzw. operacji dominujących. Operacje dominujące są określane jako te spośród realizowanych przy wykonaniu algorytmu, które będą najbardziej znaczące dla rzeczywistego czasu wykonania. Operacje dominujące mogą być różne dla różnych algorytmów w różnych dziedzinach zastosowań. W niniejszej książce obszar zainteresowań ograniczony jest do wybranej grupy algorytmów, w których operacjami dominującymi są podstawowe operacje wykonywane przez sprzęt komputerowy, takie jak operacje arytmetyczne, dostępy do pamięci, przesłania danych przez magistrale i sieci komputerowe.

Wynika to częściowo z wyboru obszaru informatyki, na którym głównie koncentruje się książka. Podstawowe opisywane algorytmy należą do numerycznej algebry liniowej i obejmują podstawowe operacje na wektorach (tablicach liczbowych) i macierzach. Operacje takie występują w wielu dziedzinach zastosowań, są między innymi podstawą bardziej rozbudowanych metod w obszarze nauk obliczeniowych (*computational science*) i obliczeń naukowo-technicznych (*scientific and technical computing*). Analiza i optymalizacja tego typu algorytmów jest związana od lat z dziedziną obliczeń wysokiej wydajności (*high performance computing*).

Zaprezentowane w książce sposoby analizy wydajności mają jednak charakter ogólny i mogą być wykorzystane w szerszym kontekście, dla algorytmów innych rodzajów, operujących na innych typach danych, wykorzystywanych w rozmaitych gałęziach informatyki. Takimi przykładowymi obszarami zastosowań, w których istotna jest wydajność przetwarzania i w których stosować można techniki omawiane w książce, są grafika komputerowa, analiza danych czy uczenie maszynowe.

Zapis implementacji algorytmów jest dokonywany w książce w języku C. Przyczyną jest z jednej strony popularność języka C, z drugiej jego podobieństwo w zapisie podstawowych konstrukcji programistycznych do innych języków programowania. Ważną cechą języka C jest także to, że wśród stosowanych obecnie języków, jest on często traktowany jako język relatywnie niskiego poziomu (bez zaawansowanych abstrakcyjnych konstrukcji), będący "blisko" sprzętu, a więc dobrze nadający się do analizy wydajności.

Programy zawarte w książce powinny pozwalać na badanie wydajności niezależnie od systemu operacyjnego i kompilatora. Na potrzeby tekstu wszystkie zostały przetestowane w środowisku systemu operacyjnego Linux, przy zastosowaniu popularnych kompilatorów *gcc* i *icc* (ten ostatni, udostępniany darmowo, po spełnieniu odpowiednich wymagań, lub płatnie przez firmę Intel, szczególnie dobrze nadaje się do badania wydajności wykonania na procesorach tej firmy). Do analizy wydajności omawianych programów wykorzystane zostały wspomagające narzędzia, dostępne powszechnie w darmowych dystrybucjach Linuxa.

Elementem badania wydajności jest także analiza kodu wykonywanego przez mikroprocesory, zapisanego w języku asemblera[1]. W książce nie występuje bezpośrednie użycie całych programów lub funkcji zapisanych w kodzie asemblera, pojawiają się natomiast specjalne wstawki do kodu źródłowego w języku C, zawierające wywołania funkcji bezpośrednio powiązanych z rozkazami procesora, odpowiednio wykorzystywane przez kompilatory (tzw. *compiler intrinsic functions*).

Książka skupia się na interakcji sprzętu i oprogramowania, na analizie kodu źródłowego i kodu asemblera programów pod kątem wydajności wykonania oraz modelowaniu wydajności. Ze względu na ograniczenia objętości przedstawione są tylko podstawowe aspekty optymalizacji, interakcji z systemem

[1]Pojęcie "asemblera" jako programu narzędziowego służącego do tworzenia kodu wykonywalnego nie jest analizowane w książce, natomiast używane zamiennie są dwa określenia zapisanego symbolicznie kodu złożonego z rozkazów procesora: asembler i (nawiązujące do wspomnianego programu narzędziowego) kod asemblera.

operacyjnym, wykorzystania bibliotek (w tym standardowych bibliotek języków programowania). Omawiane programy są najczęściej krótkimi, prostymi procedurami, które mimo to pozwalają na pokazanie wielu aspektów złożonej problematyki wydajności obliczeń.

Organizacja książki, wymagania wstępne, czytelnicy

Książka jest połączeniem podręcznika i monografii. Jako podręcznik przedstawia całościowy obraz zagadnienia, począwszy od podstaw, definiując i odpowiednio ilustrując wykorzystywane pojęcia. Jako monografia zawiera elementy zaawansowane, związane z badaniami. Dla jasności i prostoty wywodu książka często abstrahuje od szczegółów mniej istotnych dla podstawowego tematu, zbiera w całość, stosując pewne uproszczenia, informacje zawarte w wielu źródłach. Nie dokumentuje też wszystkich źródeł, wskazując tylko najważniejsze, najczęściej będące podręcznikami zawierającymi bardziej szczegółowe rozwinięcie zagadnień pokrewnych do omawianych w książce.

Zdecydowaną większość materiału zawartego w książce można odszukać w internecie (dotyczy to np. wielu zamieszczonych rysunków), w postaci wiedzy rozproszonej w rozmaitych artykułach i innych tekstach. Sensem powstania książki jest z jednej strony przedstawienie spójnego obrazu zagadnienia, traktującego temat szerzej niż pojedyncze, krótkie opracowania, a z drugiej przedstawienie go w języku polskim, co może mieć znaczenie dydaktyczne, ale także przyczyniać się do rozwijania polskiego słownictwa w dziedzinie informatyki.

Książka zakłada u Czytelnika pewien poziom wiedzy z obszarów podstaw informatyki, architektur systemów komputerowych, systemów operacyjnych, programowania w klasycznych językach proceduralnych i obiektowych oraz obliczeń równoległych. Zakres omawiany na kursach studiów informatycznych pierwszego stopnia powinien być w zupełności wystarczający. Osoby nie studiujące informatyki, a mające pewne doświadczenie programistyczne, w tym doświadczenie w programowaniu równoległym, także nie powinny mieć kłopotu z korzystaniem z książki.

Integralną częścią książki jest zestaw przykładowych zadań i problemów, omawianych i rozwiązywanych w tekście. Wybrane zadania, w formie ćwiczeń laboratoryjnych, przedstawione są także na stronie `http://www.metal.agh.edu.pl/~banas/WO/WO.html` związanej z kursem "Wydajność oprogramowania" prowadzonym przez autora. Poza omówieniem problemów oraz wykorzystywanych przy ich rozwiązywaniu sposobów analizy oraz optymalizacji wydajności, tematy ćwiczeń laboratoryjnych zawierają także szereg, pominiętych w książce, dokładnych instrukcji uruchamiania programów czy obsługi stosowanych aplikacji i narzędzi.

Książka jest na bieżąco redagowana i aktualizowana, wszelkie uwagi na jej temat można kierować na adres internetowy pobanas@cyf-kr.edu.pl .

Rozdział 2

Wybrane techniki i narzędzia wspomagające analizę wydajności

2.1 Mikrobenchmarki

Obiektem badań w książce są algorytmy i programy wykonywane na współczesnych systemach komputerowych, złożonych z mikroprocesorów, układów pamięci, rozmaitych sieci połączeniowych oraz szeregu innych elementów. Jedną z konsekwencji wysokiego stopnia złożoności samych systemów obliczeniowych oraz ich interakcji z wykonywanym oprogramowaniem, jest trudność, a niekiedy brak możliwości, uzyskania ilościowych charakterystyk wydajności i oszacowań czasu wykonania, wyłącznie na podstawie teoretycznej analizy sprzętu.

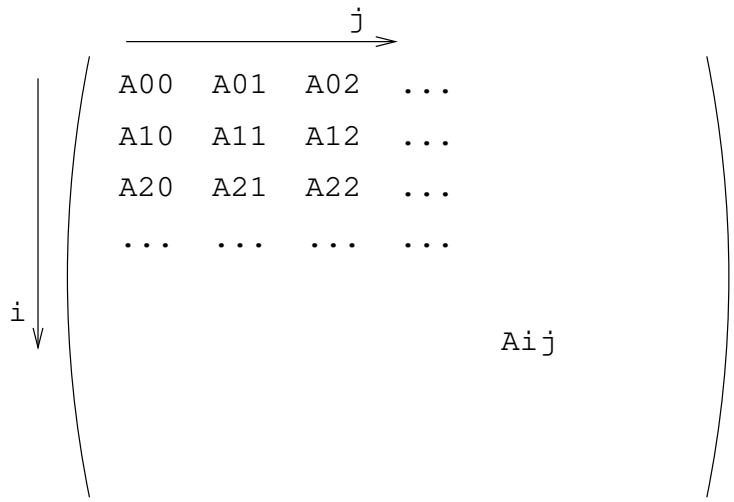
Z tego względu, często konieczne staje się wsparcie teoretycznych analiz technikami eksperymentalnymi. Powszechnie stosowanymi narzędziami w technikach eksperymentalnych są benchmarki komputerowe, specjalnie zaprojektowane programy lub ściśle określone zadania, wraz z warunkami ich wykonania przez konkretne aplikacje, których realizacja pozwala, po odpowiednich pomiarach, na obliczenie założonych miar wydajności.

Benchmarki komputerowe są stosowane do określania wydajności dla różnych form oprogramowania: grup programów użytkowych o zbliżonej charakterystyce, wybranych elementów składowych systemów informatycznych, powszechnie stosowanych funkcji bibliotecznych, czy zestawów operacji na danych. Ważnym w rozważaniach o architekturze sprzętu rodzajem benchmarków są mikrobenchmarki, służące do badania wydajności konkretnych komponentów systemów komputerowych. Zgodnie ze swą nazwą mikrobenchmarki zawierają krótkie, często kilkulinijkowe, fragmenty kodu poddawanego badaniu.

Poza testowaniem wydajności dobrze rozpoznanych elementów sprzętowych, mikrobenchmarki mogą pełnić także inną, specyficzną rolę. Często budowa i szczegóły funkcjonowania konkretnego układu nie są publicznie udostępniane przez producentów lub też wzajemne interakcje kilku układów nie są łatwe do przewidzenia dla konkretnego kodu. W takich przypadkach mikrobenchmarki można wykorzystać do wysnucia wniosków, lub postawienia hipotez, dotyczących budowy i funkcjonowania komponentów systemu oraz ich interakcji w trakcie wykonywania programów.

2.1.1 Kody mikrobenchmarków - pętle i dostępy do tablic

Kody mikrobenchmarków składają się najczęściej z prostych pętli, pojedynczych i podwójnych, zazwyczaj wykonywanych wielokrotnie. W pętlach tych realizowane są operacje arytmetyczne, z których, dla celów badań wydajności przeprowadzanych w książce, najważniejszymi będą operacje zmiennoprzecinkowe, a także dokonywane są dostępy do danych w pamięci. Dane zazwyczaj przechowywane są w



Rysunek 2.1: Macierz i oznaczenia jej elementów

postaci tablic, a czas dostępu do pojedynczej danej w trakcie wykonania, co będzie dokładnie analizowane w dalszej części książki, zależy od wzorca dostępu do pamięci w kodzie. Wzorzec może zakładać dostęp w kolejnych iteracjach pętli do kolejnych elementów tablicy, do losowych elementów tablicy, a także do elementów tablicy oddalonych w pamięci o stały odstęp, który można mierzyć liczbą bajtów, ewentualnie liczbą elementów tablicy.

Dla prostej pętli

```
for (j=0; j<rozmiar_tab; j+=skok) tab[j]++;
```

dostępy do tablicy danych `tab`, o rozmiarze `rozmiar_tab`, odbywają się począwszy od pierwszego elementu (o indeksie 0), kolejno do lokalizacji oddalonych o liczbę elementów równą wartości zmiennej `skok` (dla wyrażenia odstępu w bajtach, `skok` należy pomnożyć przez rozmiar pojedynczego elementu tablicy, np. osiem bajtów dla liczb podwójnej precyzji).

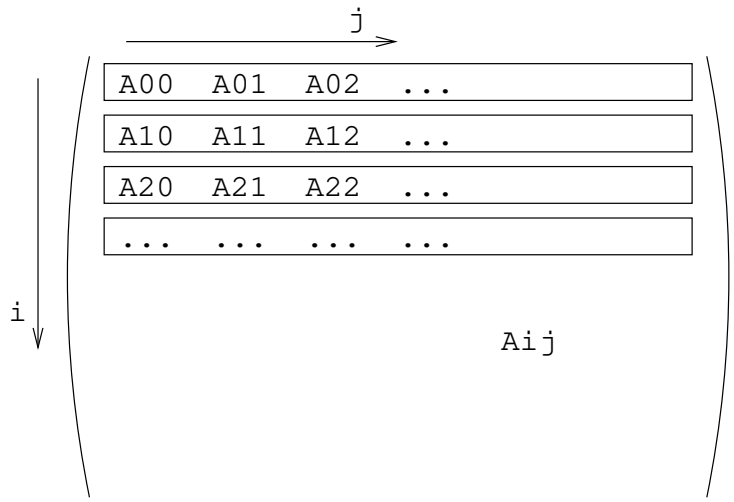
Specjalną grupę benchmarków, związanych najczęściej z numeryczną algebrą liniową, stanowią programy realizujące algorytmy z dostępami do tablic, które przechowują dane macierzy liczbowych. Macierze są zbiorami elementów, wygodnie zapisywanymi w postaci prostokątnej tablicy, które dla przykładowej macierzy $\boldsymbol{A}$ oznaczane będą jako A_{ij}, gdzie i jest indeksem wiersza, a j indeksem kolumny macierzy (rys. 2.1). W przedstawionej na rysunku konwencji, stosowanej w całej książce, indeksowanie tak wierszy, jak i kolumn, rozpoczyna się, zgodnie z tradycją C, od 0.

Najprostszym sposobem przechowywania danych macierzowych jest wykorzystanie tablic dwuwymiarowych jako struktur danych w językach programowania. Przy standardowej alokacji w C tablicy dwuwymiarowej o *M* wierszach i *N* kolumnach

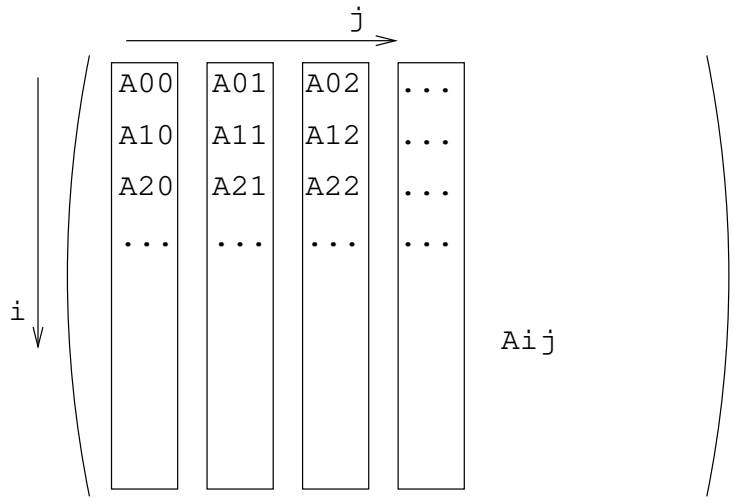
```
double A[M][N];
```

przechowywanie elementów odbywa się wierszami, co oznacza, że w kolejnych komórkach pamięci znajdują się kolejne wyrazy wiersza, odpowiadające kolejnym kolumnom, np. dla *i*-tego wiersza jego pierwsze wyrazy umieszczone są w pamięci w kolejności: $A_{i0}, A_{i1}, A_{i2}, ...$ Po umieszczeniu wiersza w pamięci, bezpośrednio po nim znajdą się wyrazy następnego wiersza (rys. 2.2). Oznacza to także, że kolejne wyrazy w dowolnej kolumnie, np. $A_{0j}, A_{1j}, A_{2j}, ...$, oddzielone będą w pamięci liczbą wyrazów równą długości wiersza *N* (będącej jednocześnie liczbą kolumn).

Przechowywanie macierzy kolumnami, jak na rys. 2.3, jest typowe dla języka Fortran i ze względu na jego popularność w obliczeniach technicznych, wciąż często stosowane w rozmaitych bibliotekach.



Rysunek 2.2: Ilustracja przechowywania macierzy wierszami



Rysunek 2.3: Ilustracja przechowywania macierzy kolumnami

Przechowywanie macierzy w tablicach jednowymiarowych

Dla większości implementacji rozważanych w książce do przechowywania elementów macierzy stosowane będą tablice jednowymiarowe. Daje to większą elastyczność w programowaniu, wprowadzając niewielkie tylko komplikacje. Przechowanie macierzy w tablicy jednowymiarowej polega na zdefiniowaniu pojedynczej tablicy o rozmiarze pozwalającym na przechowanie wszystkich wyrazów:

```
double a[M*N];
```

W technice tej możliwy jest wybór sposobu przechowywania macierzy: wierszami (*row major*) lub kolumnami (*column major*). Sposób przechowywania wierszami, odpowiadający standardowej praktyce języka C, oznacza, że wyraz A_{ij} tablicy $\boldsymbol{A}$ znajduje się w elemencie o indeksie `a[i*N+j]` (*i*-ty wiersz, *j*-ty element w wierszu $=$ *j*-ta kolumna). Przechowywanie kolumnami będzie polegało na przypisaniu wyrazowi A_{ij} miejsca w pamięci `a[i+j*N]`.

Przeglądanie macierzy zapisanych w postaci tablic jednowymiarowych nadal najczęściej realizowane jest w podwójnej pętli, po wierszach i kolumnach. Dla tablic przechowywanych wierszami, przeglądanie z pętlą po wierszach jako zewnętrzną:

```
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
    ... a[i*N+j] ... // A[i][j]
  }
}
```

polega na odwiedzaniu kolejnych komórek w pamięci. Przeglądanie z pętlą po kolumnach jako zewnętrzną:

```
for(j=0;j<N;j++){
  for(i=0;i<M;i++){
    ... a[i*N+j] ... // A[i][j]
  }
}
```

oznacza odwiedzanie w kolejnych iteracjach wyrazów oddalonych o *N* elementów, a więc kolejne dostępy do pamięci ze skokiem np. *8*N* bajtów dla zmiennych podwójnej precyzji.

W praktycznych przykładach w książce używane są zawsze macierze kwadratowe, najczęściej występujące w zastosowaniach naukowo-technicznych. W macierzach takich liczba wierszy i kolumn jest równa i określana jest jako wymiar macierzy, najczęściej oznaczany przez N. W sytuacji takiej rozmiar macierzy wynosi NxN.

Array padding czyli rozciąganie tablic

Dostęp do komórek pamięci w kolejnych iteracjach pętli ze skokiem o pewną liczbę bajtów może powodować opóźnienie działania programu, szczególnie widoczne dla specyficznych rozmiarów skoku. Może zdarzyć się tak, że skok przy dostępie do tablicy w algorytmie operującym na macierzach zależy od wymiarów macierzy (np. w opisanym powyżej dostępie do kolejnych elementów w pojedynczej kolumnie, w przypadku przechowywania macierzy wierszami).

Popularną techniką zapobiegania ewentualnym przyrostom czasu wykonania dla specyficznych rozmiarów macierzy jest technika *array padding*, co w dalszej części książki będzie określane jako rozpychanie lub rozciąganie tablic. Polega ona na zaalokowaniu tablic większych niż wymagane w algorytmie, co pozwala między innymi na uniknięcie niekorzystnego wzorca dostępów do pamięci (*array padding* można stosować także w innych optymalizacjach, niekoniecznie związanych z korzystaniem z pamięci).

Dla tablic dwuwymiarowych zamiast alokować

```
double A[M][N];
```

definiuje się tablicę o wydłużonym wierszu:

```
double A[M][N+O];
```

gdzie parametr `O` dobierany jest odpowiednio do wymagań optymalizacji. Algorytmy modyfikuje się tak, żeby niezależnie od rozmiaru zaalokowanej pamięci dotyczyły tylko wyrazów odpowiadających oryginalnym tablicom, ewentualnie wypełnia się powiększone tablice, tak aby operując na dodanych wyrazach nie powodować zmiany wyników algorytmu.

W przypadku stosowania tablic jednowymiarowych, alokacja w technice *array padding* dotyczy `M*(N+O)` wyrazów, a dostęp do wyrazu A_{ij} macierzy przechowywanej wierszami w tablicy `a` (w *i*-tym wierszu i *j*-tej kolumnie), uzyskuje się za pomocą notacji `a[i*(N+O)+j]`.

2.2 Narzędzia wspomagające analizę wydajności

W książce wykorzystywany jest szereg programów narzędziowych typowych dla dziedziny badania wydajności. Podstawowymi narzędziami są tzw. *profilery*, służące do uzyskiwania czasów wykonania dla całych programów oraz ich poszczególnych fragmentów. Często pozwalają one także na uzyskanie dodatkowych informacji o zdarzeniach związanych z wykonaniem programu, mających istotny wpływ na wydajność (np. liczby dostępów do różnych rodzajów pamięci).

Profilery mogą działać w oparciu o kilka różnych zasad. Jedną z nich jest tzw. *instrumentacja* kodu, polegająca na dodaniu do programu, najczęściej w trakcie kompilacji, fragmentów przekazujących informację o zdarzeniach (*events*) związanych z wykonaniem kodu (np. wywołanie funkcji, przejście w tryb jądra systemu operacyjnego, itp.).

Inną z zasad działania profilerów jest statystyczne próbkowanie (*statistical sampling*). W trakcie wykonania programu (z kodem wykonywalnym poddanym wcześniejszej instrumentacji, ewentualnie w przypadku realizacji programu pod kontrolą odpowiedniego środowiska nadzorującego), w określonych odstępach czasu, przetwarzanie jest przerywane i odczytywane są dane mające znaczenie dla analizy wydajności. Pobierane mogą być dane różnego rodzaju, np. stan stosu wywołań (*execution stack, call stack*) lub zawartość specjalnych rejestrów procesora (tzw. liczników sprzętowych, *hardware counters*), zliczających wskazane zdarzenia dotyczące użycia sprzętu (*hardware events*), takie jak np. dostępy do pamięci, takty zegara, wykonane rozkazy (bardziej szczegółowy opis wykorzystania liczników sprzętowych znajduje się w p. 3.9). Ze względu na statystyczną naturę zbierania danych, gdzie wnioskuje się o całości wykonania programu na podstawie próbkowania w wybranych chwilach czasu, wyniki zwracane przez tego typu profilery bywają obarczone błędami. Raportowana liczba zdarzeń może być różna od rzeczywistej (zazwyczaj w granicach kilku, kilkunastu procent), pewne rzadziej występujące zdarzenia mogą zostać pominięte.

Nadzorowanie wykonania programu może przybierać różne formy. Od prostego zbierania danych dotyczących zdarzeń lub uzyskanych z próbkowania, aż do pełnienia funkcji maszyny wirtualnej, która staje się środowiskiem wykonania programu. W takim przypadku, program nadzorujący, wykonujący pojedyncze instrukcje kodu, uzyskuje możliwość precyzyjnego badania wszystkich zdarzeń podczas wykonania. Wadą takiego podejścia jest występujący najczęściej duży narzut czasowy związany z realizacją obliczeń w ramach takiej maszyny wirtualnej. Narzut ten może być relatywnie niewielki w przypadku kiedy program (np. w postaci odpowiedniego kodu pośredniego, często poddawanego uprzednio instrumentacji) standardowo wykonywany jest przy pomocy maszyny wirtualnej, np. dla maszyn wirtualnych Javy lub .NET.

Informacje zbierane w trakcie wykonania mogą być na bieżąco udostępniane przez program nadzorujący, np. w postaci graficznej, lub zapisywane w odpowiednich plikach. Dane z plików mogą być odczytywane, interpretowane i wizualizowane przez dowolne programy rozpoznające format zapisu.

Istnieją dwie podstawowe formy prezentacji danych związanych z wykonaniem kodu, w szczególności dotyczących wydajności. Jedną z nich jest tzw. profil wykonania (*execution profile*), a drugą ślad wykonania (*execution trace*). Profil wykonania to zbiorcze zestawienie wybranych danych zebranych w trakcie wykonania, w szczególności czasów realizacji poszczególnych fragmentów kodu: funkcji, bloków kodu, pojedynczych instrukcji. Dane mogą być wzbogacone o informacje związane z grafem wywołań (np. ile razy dana funkcja wywoływana była przez inną wybraną funkcję).

Ślad wykonania to zapis zdarzeń w kolejności chronologicznej (rozmiar pliku śladu, w przeciwieństwie do rozmiaru pliku profilu, rośnie proporcjonalnie do czasu działania programu). W badaniach wydajności, ślady wykonania są szczególnie popularne przy wykonaniu w środowiskach przesyłania komunikatów, gdzie pozwalają na optymalizację sposobu i czasu komunikacji między procesami.

W książce stosowane są narzędzia zbierania danych o wykonaniu powszechnie dostępne dla darmowych dystrybucji Linuxa. Najpopularniejszym z narzędzi jest program *gprof*. Wykorzystuje on hybry-

dowy mechanizm, łącząc próbkowanie statystyczne z instrumentacją kodu (*gprof* wymaga kompilacji ze specjalnymi opcjami, standardowo *-p* lub *-pg*). *gprof* tworzy tzw. płaski profil (*flat profile*) oraz profil z drzewem wywołań (*call graph*). Płaski profil zawiera informacje o czasie wykonania poszczególnych funkcji kodu (bezwzględnym oraz procentowym w stosunku do czasu wykonania całego programu), w tym także ile czasu zajmowało wykonanie samej funkcji, bez innych wywoływanych przez nią procedur, a ile łącznie z nimi. Profil z drzewem wywołań rozróżnia dla każdej funkcji czasy jej wykonania zależnie od procedury wywołującej tę funkcję. Użycie narzędzia *gprof* jest często pierwszym krokiem optymalizacji kodu – służy do wykrycia tych procedur, w których program spędza najwięcej czasu, a więc optymalizacja których może przynieść największe zyski czasowe.

Inne z wykorzystywanych, bardziej specjalistycznych narzędzi analizy wydajności programów, omówione są w późniejszych rozdziałach książki, w miejscach ich bezpośredniego zastosowania (np. narzędzie *perf* w p. 3.9.1, biblioteka PAPI w p. 3.9.1, narzędzie *valgrind* w p. 5.3.1)

Wydajność efektywna

Skutkiem ubocznym zastosowania pewnych technik optymalizacji, w tym także np. *array padding*, może być sytuacja, kiedy w implementacji pewnego algorytmu wykonuje się dodatkowe operacje lub dostępy do pamięci, nie występujące w algorytmie i nie wpływające na jego użyteczne wyniki (w przypadku rozciągania tablic może to dotyczyć dostępów do i operacji na dodanych elementach wiersza o indeksach poza zakresem jego oryginalnej długości).

Jeśli wydajność programu wyrażana będzie w liczbie operacji na sekundę lub liczbie dostępów na sekundę (co może być także tłumaczone na szybkość transferu danych z pamięci), można przyjąć konwencję uwzględniania tylko tych operacji i dostępów, które występują w oryginalnym algorytmie, tzn. takich, które wykonują użyteczną pracę, ze względu na wynik obliczeń.

Taki sposób liczenia wydajności przyjmowany będzie we wszystkich badanych w książce programach użytkowych. Wyjątkiem od tej reguły będą pewne mikrobenchmarki, ukierunkowane wyłącznie na badanie funkcjonowania samego sprzętu.

Zastosowane podejście, uwzględniania wyłącznie operacji użytecznych, a nie wszystkich wykonanych przez sprzęt, jest odpowiednie nie tylko w przypadku wspomnianych technik optymalizacji (które mogą wprowadzać nieużyteczne, dodatkowe operacje, w celu skrócenia czasu wykonania programu), ale wynika także np. z analizy pracy procesorów (rdzeni). Często wykonują one wiele operacji nie odpowiadających instrukcjom kodu (i rozkazom asemblera), np. przy stosowaniu, omawianych w dalszych częściach książki, technik wykonania spekulatywnego (*speculative execution*), takich jak np. pobieranie z wyprzedzeniem (*prefetching*) czy przewidywanie skoków (*branch prediction*). Z przyjęcia założenia, że wydajność dotyczy tylko operacji efektywnie wykonanych na potrzeby aplikacji, w pewnych przypadkach może wynikać także względna przydatność zliczania zdarzeń sprzętowych – w analizie istotne jest nie tyle ile operacji, zliczanych przez liczniki sprzętowe, wykonał procesor (rdzeń), ale ile z tych operacji przełożyło się na efektywną pracę programu.

Rozdział 3

Model przetwarzania i wydajność obliczeń kodu sekwencyjnego na pojedynczym rdzeniu mikroprocesora

3.1 Mikroarchitektury – architektury rdzeni mikroprocesorów

Ze względu na ogromny stopień złożoności systemów obliczeniowych, spojrzenie na ich architekturę jest w książce z konieczności uproszczone i skupia się na wybranych elementach, których wpływ na wydajność obliczeń daje się prześledzić na poziomie analizy kodu źródłowego oraz sterowania parametrami wykonania programów.

Zawarty w niniejszym punkcie opis mikroarchitektury, dotyczy podstawowych cech architektury pojedynczego rdzenia mikroprocesora, odpowiadającego klasycznej jednostce przetwarzania pojedynczego wątku obliczeń (pojedynczej sekwencji rozkazów). Jednostka taka była określana tradycyjnie jako procesor (*CPU, central processing unit*), stąd też nazwy rdzeń i procesor często w książce używane są zamiennie (czasem stosowana jest także nazwa procesor logiczny)[1].

Prezentacja architektury pojedynczego rdzenia mikroprocesora, podobnie jak opisy innych elementów sprzętowych, jest gdzieniegdzie dokonywana na poziomie elementarnym. Ma to między innymi za zadanie wprowadzenie i jednoznaczne zdefiniowanie szeregu podstawowych pojęć stosowanych w dalszej części książki przy analizie i optymalizacji wydajności wykonania programów.

3.2 Architektura von Neumanna

Podstawowym modelem mikroarchitektury pojedynczego rdzenia, będącym punktem wyjścia analiz i realizowanym (z ewentualnymi modyfikacjami) we wszystkich współczesnych systemach komputerowych, jest model architektury von Neumanna. Sposób modyfikacji i rozbudowy jej podstawowych elementów definiuje szereg istotnych różnic między współczesnymi mikroarchitekturami.

Maszyna von Neumanna, jako podstawowy model obliczeniowy, składa się z jednostki centralnej (CPU, *Central Processing Unit*), połączonej za pomocą odpowiednich kanałów komunikacyjnych z układem adresowalnej binarnej pamięci głównej (*main memory*). Procesor realizuje program zapisany w pamięci głównej, korzystając z danych wejściowych, przechowywanych w tej samej pamięci, i zapisując wyniki obliczeń, także w pamięci głównej.

[1] Pojedyncze układy scalone mające postać mikroprocesorów wielordzeniowych, będą w książce czasami także skrótowo nazywane procesorami, w sytuacjach kiedy kontekst jawnie wskazuje, które znaczenie określenia procesor jest właściwe.

Adresowalność pamięci oznacza, że można podzielić pamięć główną na podstawowe elementy, zawierające bity informacji, nazywane dalej komórkami pamięci (*memory cell, memory location*). Każda z takich komórek posiada adres, dzięki któremu możliwe jest pobieranie i zapisywanie danych w dowolnej komórce. Rozmiar (w bitach) pojedynczej (czyli posiadającej jeden adres) komórki pamięci może być różny, we wszystkich przykładach wykorzystywanych w książce wynosi jeden bajt (będący standardową jednostką dla najważniejszych współczesnych mikroprocesorów).

W pamięci przechowywane są dane o różnym typie, takie jak znaki, liczby całkowite, liczby zmiennoprzecinkowe. Pojedyncze zmienne określonych typów mogą być reprezentowana za pomocą różnej liczby bajtów. Stąd, w niniejszej książce, pojęcia adresu i komórki pamięci są często używane w znaczeniu mniej ścisłym, a bardziej ogólnym: często stosowane będą określenia: "komórka przechowująca liczbę" (choć w rzeczywistości liczba może być przechowywana w kilku komórkach), "adres zmiennej" (co oznaczać będzie adres pierwszej komórki, w której znajdują się bity danej liczby).

Program w pamięci głównej składa się z ciągu przechowywanych rozkazów (*instructions*). Wykonanie programu przez procesor polega na pobieraniu kolejnych wykonywanych rozkazów (kolejny wykonywany niekoniecznie oznacza kolejny przechowywany w pamięci, np. przy realizacji rozkazu skoku), a następnie ich przetwarzaniu. Perspektywa przyjęta w niniejszej książce, związana z analizą wydajności wykonania, preferuje zawsze spojrzenie na wykonywane rozkazy, a nie rozkazy zapisane w kodzie binarnym. Wydajność staje się istotna w przypadku wykonywania wielkiej liczby rozkazów (rzędu co najmniej miliardów), co zawsze przekracza liczbę rozkazów w kodzie binarnym. W skrajnych przypadkach, przy użyciu pętli o bardzo dużej liczbie iteracji, analizowany kod binarny może zawierać tylko kilka lub kilkanaście rozkazów. W praktyce oznacza to, że często przy analizie wydajności obiektem zainteresowania sa tylko wybrane, często krótkie fragmenty kodu, które jednak prowadzą do dużej liczby realizowanych rozkazów i znacznego czasu wykonania (tzw. punkty zapalne, *hotspots*).

Podstawowymi rozkazami, uwzględnianymi w niniejszej książce, wykonywanymi przez procesory są:

- pobieranie z pamięci i zapis do pamięci
- operacje arytmetyczne
- operacje logiczne
- transfer sterowania, skoki
- operacje wejścia/wyjścia

Rozkazy, jak widać z powyższego zestawienia, zawsze oznaczają wykonanie pewnej operacji, która może posiadać od zera do kilku argumentów (zazwyczaj maksymalnie trzech). Argumentami rozkazów mogą być liczby (argumenty bezpośrednie), zawartość rejestrów, traktowanych jako komórki wewnętrznej pamięci procesora o określonym rozmiarze i indywidualnych nazwach, oraz zawartość komórek pamięci, dostępna dzięki adresowi przechowywanemu w rejestrach.

W książce dla ilustracji zagadnień wydajności obliczeń, pojawiać się będą przykłady kodu, realizowanego przez procesory, zapisanego w wersji języka asemblera typowej dla systemów operacyjnych z rodziny Unix. Oznaczeniami typowych, przykładowych rozkazów (występujących np. w procesorach rodziny x86) są:

- `mov` : przesunięcie danych pomiędzy rejestrami i komórkami pamięci (bez operacji pamięć-pamięć)
- `ld, st` : dostępy do pamięci – pobranie i zapis
- `add, sub, mul` : dodawanie, odejmowanie, mnożenie argumentów

- `inc, dec` : zwiększenie, zmniejszenie o 1
- `neg` : zmiana znaku
- `lea` : obliczenie adresu (bez transferu danych)
- `xor, and, or` : działania logiczne (na bitach argumentów)
- `cmp` : obliczenie wartości logicznej porównania dwóch argumentów i zapisanie wyniku w odpowiednich rejestrach procesora (rejestrach stanu)
- `jmp` : skok bezwarunkowy
- `jge, je, jl` : skoki warunkowe – przeniesienie sterowania do określonego miejsca kodu, zależnie od wyniku poprzedzajacej operacji porównania, zapisanego w rejestrze stanu
- `call, ret` : obsługa wywołań procedur

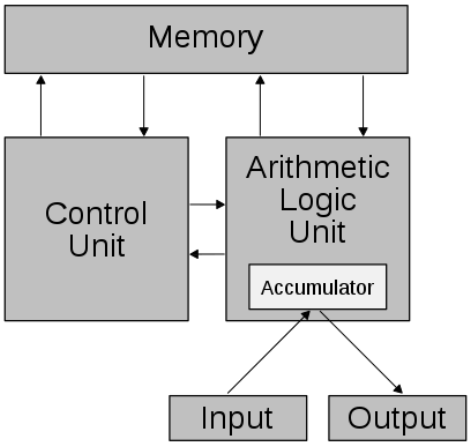
W ramach przyjętej notacji, zawartości rejestrów (najczęściej stosowane będą rejestry 32 i 64-bitowe oraz wektorowe) zapisywane są ze znakiem `%` (np. `%eax` jako zawartość 32-bitowego rejestru `eax`), a zawartości komórek pamięci głównej z użyciem nawiasów (np. `(%rax)` jako zawartość komórek pamięci o adresie początkowym zapisanym w 64-bitowym rejestrze `rax`). Liczba komórek pamięci, których dotyczy rozkaz, jest zależna od konkretnego rozkazu, czasem nazwa rozkazu jest modyfikowana w celu określenia rozmiaru argumentu.

W rozkazach procesora często stosuje się złożone tryby adresowania, szczególnie wygodne przy dostępie do elementów tablic, gdzie pojedynczy adres obliczany jest na podstawie kilku wartości. Dla zapisu *disp(base, index, scale)* (w przyjętej w pracy notacji asemblera $x86$) obliczenie adresu ma postać: *adres = base + index*scale + disp*, a wykorzystywanymi wartościami są:

- baza (*base*) adresu (zawarta w odpowiednim rejestrze)
- indeks (*index*), zapisany także w rejestrze, służący często jako reprezentacja indeksu elementu w tablicy
- współczynnik skalowania (*scale*), wykorzystywany do mnożenia przez wartość indeksu, przy dostępie do tablic oznaczający rozmiar elementu tablicy (odstęp pomiędzy kolejnymi elementami) w bajtach, równy 1, 2, 4 lub 8
- przesunięcie (*disp*), używane w sytuacji kiedy obliczany adres jest oddalony o *disp* bajtów od bazy i nie jest związany z dostępem do tablic (ten sposób adresowania jest powszechnie stosowany dla zmiennych na stosie)

.

Specjalne układy procesora służące do obliczania adresów w powyższej formie są udostępniane także dla rozkazów, w których nie dokonuje się dostępu do pamięci (np. `lea`, *load effective address*, obliczający adres i zapisujący go w rejestrze). Rozkazy te bywają wykorzystywane przez optymalizujące kompilatory do wykonywania czysto arytmetycznych operacji, np. zawierających mnożenie przez 2, 4 lub 8.



Rysunek 3.1: Architektura von Neumanna [źródło: Wikipedia]

3.3 Przetwarzanie rozkazów

Przetwarzanie pojedynczego rozkazu składa się z szeregu etapów, o liczbie i charakterze zależnym od konkretnych rozwiązań technicznych procesora, wśród których zawsze można jednak wyróżnić podstawowe fazy:

- pobrania rozkazu z pamięci do procesora
- dekodowania rozkazu (w uproszczonym ujęciu można to rozumieć jako przekształcenie pobranego rozkazu na sekwencję wewnętrznych operacji procesora, czasem dekodowanie rozkazu z listy rozkazów procesora związane jest z jego zamianą na sekwencję realizowanych mikro-rozkazów)
- wykonania rozkazu

Wykonanie rozkazu może być związane z pobraniem argumentów rozkazu z rejestrów lub pamięci głównej, zapisem wyniku lub zmianą wewnętrznego stanu procesora.

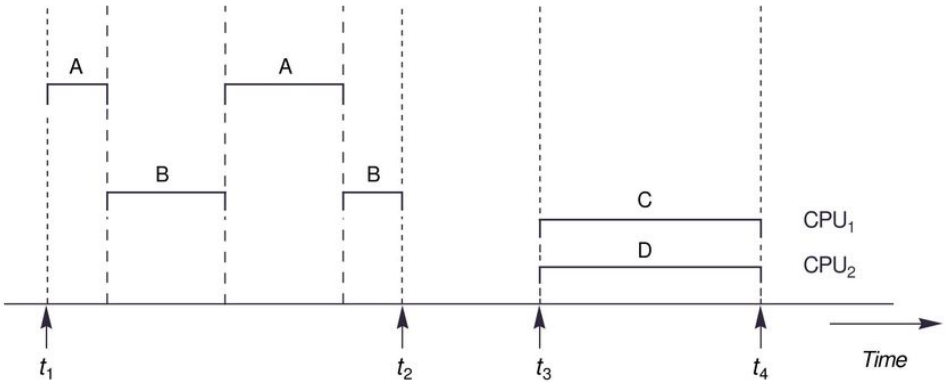
W klasycznej maszynie von Neumanna całość układu przetwarzania można schematycznie przedstawić jak na rys. 3.1. Poza pamięcią (*Memory*) i procesorem, składającym się z jednostki sterującej (*Control Unit)* oraz jednostki wykonania rozkazów (*Arithmetic-Logic Unit*) z wyróżnionym pojedynczym rejestrem (*Accumulator*), schemat obejmuje także urządzenia wejścia/wyjścia (*Input/Output*).

3.4 Jednostki wykonania rozkazów

Rozszerzenia i modyfikacje pierwotnej architektury von Neumanna obecne we współczesnych mikroprocesorach obejmują bardzo szeroki zakres, tak jeśli chodzi o liczbę elementów składowych procesora, jak i o stopień ich złożoności. Pierwszym z analizowanych elementów jest układ wykonywania rozkazów.

3.4.1 Przetwarzanie potokowe

Wykonanie rozkazów przez współczesne procesory odbywa się zawsze z wykorzystaniem potoków przetwarzania rozkazów (*instruction pipeline*). Przetwarzanie pojedynczego rozkazu jest rozbijane na etapy, każdy z etapów wykonywany jest przez odrębne układy, dzięki czemu procesor współbieżnie



Rysunek 3.2: Wykonanie współbieżne: w przeplocie (procesy/wątki A i B) oraz równoległe (procesy/wątki C i D) [źródło: Wikipedia]

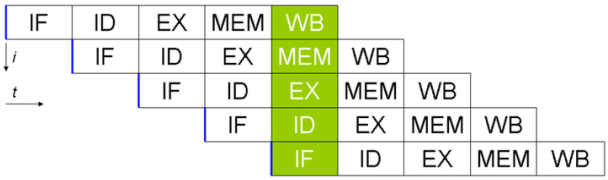
Rysunek 3.3: Schemat przetwarzania bez wykorzystania współbieżności [źródło: Wikipedia]

przetwarza kilka rozkazów. W dalszej części książki współbieżność zawsze będzie oznaczała realizację wielu działań (w tym przypadku przetwarzanie wielu rozkazów, dalej także wykonanie wielu wątków lub procesów), w taki sposób, że rozpoczęcie realizacji kolejnego działania rozpoczyna się przed zakończeniem poprzedniego. Współbieżność może mieć wiele postaci, począwszy od realizacji wielu działań w przeplocie (kiedy w konkretnej chwili realizowane jest tylko jedno działanie), poprzez realizacje potokową, kiedy równocześnie może być realizowane kilka działań, ale każde znajduje się w innej fazie, aż po pełną równoległość, kiedy wiele jednocześnie realizowanych działań może znajdować się w tej samej fazie (rys. 3.2).

Przyjmując podział przetwarzania rozkazu przez procesor na przykładowe etapy (przy czym w rzeczywistości, różne rozkazy mogą mieć różne etapy – czasem jest ich więcej, czasem mniej niż w przedstawionym przykładzie): IF – pobranie rozkazu (*instruction fetch*), ID – dekodowanie rozkazu (*instruction decode*), EX – wykonanie operacji składających się na rozkaz (*instruction execute*), MEM – dostęp do pamięci (*memory access*) oraz WB – zapis efektu realizacji rozkazu (*write-back*), schemat klasycznego przetwarzania sekwencyjnego rozkazów można zilustrować jak na rys. 3.3.

Wykorzystanie podziału procesora na pracujące równolegle podukłady związane z realizacją poszczególnych faz przetwarzania rozkazu prowadzi do przetwarzania potokowego pokazanego na rys. 3.4. Widoczne jest, że procesor w jednej chwili czasu (biegnącego wzdłuż osi poziomej) przetwarza kilka rozkazów, z których każdy znajduje się w innej fazie (na rysunku pojedynczy rozkaz złożony z przykładowych faz jest pojedynczym poziomym blokiem, a pionowy zielony blok odpowiada przykładowemu odcinkowi czasu - np. pojedynczemu taktowi procesora).

Porównując z przetwarzaniem nie wykorzystującym współbieżności (rys. 3.3), widać zyski czasowe związane z przetwarzaniem potokowym. Jeśli przyjmiemy, że każdy etap przetwarzania potokowego zajmuje ten sam odcinek czasu t_e, to przetwarzanie rozkazu o k etapach zajmuje $k \cdot t_e$ czasu. Dla sekwencji n rozkazów daje to w przypadku nie wykorzystywania współbieżności czas wykonania $n \cdot k \cdot t_e$. W przypadku przetwarzania potokowego, czas wykonania pierwszego rozkazu to $k \cdot t_e$, podczas gdy ukończenie pozostałych $n-1$ rozkazów zajmuje dodatkowo $(n-1) \cdot t_e$, co ostatecznie daje czas wykonania



Rysunek 3.4: Schemat klasycznego przetwarzania potokowego [źródło: Wikipedia]

$(n + k - 1) \cdot t_e$. Dla odpowiednio długiej sekwencji rozkazów, $n \gg k$, skrócenie czasu wykonania związane z zastosowaniem potokowości, $\frac{n \cdot k \cdot t_e}{(n+k-1) \cdot t_e}$, zmierza więc do k.

W praktyce czas wykonywania rozkazów wyraża się często w taktach (cyklach) zegara procesora (*clock cycle*). Często założeniem jest takie zaprojektowanie procesora, aby wykonanie jednego etapu przetwarzania zajmowało jeden takt zegara. Dzięki temu w przetwarzaniu potokowym możliwe staje się uzyskanie sytuacji, w której po każdym takcie zegara kończone jest przetwarzanie jednego rozkazu procesora (jak np. można interpretować rys. 3.4). Powyższe założenie będzie dalej standardowo przyjmowane przy analizie pracy potoków przetwarzania w rdzeniach.

Przetwarzanie potokowe ma jeszcze inny aspekt istotnie wpływający na wydajność. Staranne zaprojektowanie potoków przetwarzania (z kilkunastoma lub kilkudziesięcioma etapami) umożliwia znaczne zwiększenie częstotliwości pracy procesorów. W taki sposób wprowadzane w latach 80-tych XX wieku procesory z rodziny RISC (*Reduced Instruction Set Computers*, procesory o zredukowanej liście uproszczonych, ale dobrze dostosowanych do przetwarzania potokowego, rozkazów) wypierały, dzięki wyższej częstotliwości i wydajności pracy, dawniejsze architektury, nazwane później procesorami CISC (*Complex Instruction Set Computers*).

Rozróżnienie procesorów na rodziny CISC i RISC straciło współcześnie na znaczeniu – mikroprocesory posiadające rozkazy typowe dla CISC na swojej liście rozkazów (np. mikroprocesory z rodziny *x86*), zazwyczaj wewnętrznie transformują je na sekwencję (mikro-)rozkazów typowych dla RISC.

3.4.2 Miary wydajności - opóźnienie i przepustowość przetwarzania potokowego

Z przetwarzaniem rozkazów przez procesor związana jest jedna z klasycznych miar wydajności – liczba taktów na rozkaz (*CPI, cycles per instruction*). Ideą wprowadzenia tej miary jest chęć szacowania czasu wykonania programu (liczonego w taktach procesora), jako sumy wartości CPI poszczególnych rozkazów w programie (lub iloczynu liczby wykonanych rozkazów i miary CPI, przy założeniu, że wykonanie każdego rozkazu wymaga tej samej liczby taktów).

W pierwotnych ujęciach, bez uwzględnienia przetwarzania potokowego, miarę CPI można było wiązać z liczbą taktów wymaganych do realizacji pojedynczego, izolowanego rozkazu. Tak definiowany czas wykonania odpowiada pojęciu opóźnienia (zwłoki, *latency*). Ogólnie, opóźnienie związane z wykonaniem konkretnej operacji można definiować jako czas od rozpoczęcia realizacji operacji do jej zakończenia (w dalszej części książki pojecie opóźnienia stosowane będzie do różnych komponentów systemów komputerowych i różnych realizowanych operacji).

W przypadku przetwarzania zilustrowanego na rys. 3.3, można przyjąć, że opóźnienie każdego z rozkazów wynosi k taktów zegara (zgodnie z założeniem, że przetwarzanie pojedynczego etapu zajmuje jeden takt). Zastosowanie miary CPI powiązanej z opóźnieniem prowadziłoby do jej wartości równej k i poprawnego czasu wykonania n rozkazów: $n \cdot k$. Jednak przyjęcie CPI równego k prowadzi do błędnego oszacowania czasu wykonania dla przypadku przetwarzania potokowego z rys. 3.4. Zamiast poprawnego czasu $n + k - 1$ uzyskuje się znacznie wyższą wartość $n \cdot k$, zaniżającą wydajność przetwarzania.

Ze względu na fakt, że współczesne procesory odrębnie dokonują pobierania i dekodowania rozkazów, a odrębnie ich wykonania, opóźnienie przy przetwarzaniu rozkazów określa się zazwyczaj tylko

w odniesieniu do faz wykonania (np. EX, MEM, WB). Wynosi ono zazwyczaj kilka taktów, choć dla złożonych rozkazów może wynosić kilkanaście lub nawet kilkadziesiąt (dane na temat opóźnienia dla konkretnych rozkazów można znaleźć w podręcznikach programowania i optymalizacji dla konkretnych mikroprocesorów). Nadal jednak powiązanie miary CPI z tak definiowanym opóźnieniem nie nadaje się do szacowania czasu wykonania w przypadku przetwarzania potokowego, wciąż znacząco zaniżając wydajność.

Poprawne oszacowanie można uzyskać analizując przypadek z rys. 3.4. Widać, że dla odpowiednio dużej liczby rozkazów istotna dla czasu wykonania jest liczba taktów, jaka mija pomiędzy chwilami zakończenia kolejnych rozkazów. Gdyby za miarę CPI przyjąć tę właśnie liczbę, czas wykonania w taktach rzeczywiście byłby równy iloczynowi CPI i liczby rozkazów n. Uzyskana liczba taktów dla przypadku z rys. 3.4, $1 \cdot n$ (przyjmując, że CPI wynosi 1), dla odpowiednio długiej sekwencji rozkazów, $n \gg k$, dobrze przybliża wartość dokładną $n + k - 1$, z dokładnością do kilku $(k - 1)$ początkowych taktów.

W efekcie, przy uwzględnieniu przetwarzania potokowego, szacowanie czasu wykonania przestaje być powiązane z opóźnieniem pojedynczego rozkazu, a staje się zależne od możliwości jak najefektywniejszego zrealizowania przez sprzęt dużej liczby operacji. Możliwości te określane są za pomocą miar przepustowości (*throughput*) wykonywania operacji, definiowanych wprost jako stosunek liczby wykonanych operacji do czasu wykonania. Przepustowość określana jest zazwyczaj dla wykonywania nieskończonego strumienia operacji lub strumienia wystarczająco dużej liczby operacji, pozwalającej pominąć opóźnienia związane z pojedynczą operacją. Często także, określenia "przepustowość" używa się w przypadku optymalnych warunków przetwarzania, kiedy sprzęt uzyskuje swoją maksymalną wydajność.

Przetwarzanie potokowe jest jedną z technik ukrywania opóźnienia (*latency hiding*) – zwiększania wydajności przetwarzania (przepustowości) bez konieczności zmniejszania opóźnienia pojedynczej operacji. Ukrywanie opóźnienia pojawia się w rozmaitych dziedzinach techniki, także techniki obliczeniowej, gdzie w trakcie wykonywania dużej liczby operacji usiłuje się uzyskać czas realizacji krótszy niż wynikający z sumowania opóźnień pojedynczych operacji.

Przyjęcie miary CPI równej 1 dla przykładowego przetwarzania potokowego z rys. 3.4 odpowiada sytuacji idealnej, kiedy nic nie zaburza przetwarzania, a procesor w żadnej chwili nie jest zmuszony do wstrzymania realizacji któregokolwiek ze współbieżnie wykonywanych rozkazów. Takie idealne sytuacje, rzadko obserwowane w praktyce, służą często do określania maksymalnych wydajności sprzętu.

Jedną z miar, które można wykorzystać w tym celu jest miara IPC, liczba rozkazów na takt (*instructions per cycle*), która definiowana jest jako liczba rozkazów kończonych w każdym takcie przez procesor [2]. Dla różnych procesorów można próbować oszacować maksymalne wartości IPC, związane z konkretnymi typami rozkazów. Popularne jest określanie maksymalnej liczby operacji zmiennoprzecinkowych, których wykonanie może skończyć procesor w każdym takcie. Liczba ta jest związana z liczbą i charakterem potoków przetwarzania operacji zmiennoprzecinkowych i służy do określania maksymalnej wydajności (przepustowości) wykonywania operacji zmiennoprzecinkowych przez procesor.

Rozwiązaniem pośrednim pomiędzy miarami związanymi z opóźnieniem przetwarzania rozkazów (zaniżającymi zazwyczaj wydajność) i maksymalną przepustowością przetwarzania (najczęściej zawyżającą wydajność) może być określanie miar wydajności w oparciu o rzeczywiste parametry wykonania programu. Takie miary można definiować jako przeciętne, uśrednione wartości w czasie realizacji obliczeń. W dalszej części książki obie miary IPC i CPI oznaczać będą takie właśnie miary uśrednione. IPC

[2] Dla skrócenia opisu stosowane będą określenia "wydanie rozkazu" (*instruction issue*) jako rozpoczęcie fazy wykonania rozkazu przez procesor (po pobraniu, zdekodowaniu i ewentualnych innych wstępnych operacjach) oraz "kończenie rozkazu" (*instruction retirement*) jako kończenie realizacji rozkazu (skończony rozkaz, *retired instruction*, to rozkaz, którego realizacja została zakończona, przy czym zakończenie może obejmować dodatkowe operacje po opuszczeniu potoków przetwarzania, np. przemianowanie rejestrów).

definiowane będzie jako iloraz liczby zrealizowanych rozkazów (l^{ins}) przez liczbę taktów zegara, jaką zajęło wykonanie rozkazów (c^{ins}), dając w efekcie, dla konkretnej sekwencji wykonanych rozkazów, liczbę rozkazów kończonych przeciętnie w pojedynczym takcie zegara. W przykładzie z rys. 3.4 daje to wartość IPC równą $\frac{n}{(n+k-1)} \to 1$, dla $n \to \infty$.

Uśrednione miary uzyskiwane są, jak widać z powyższego wzoru, eksperymentalnie, na podstawie pomiaru czasu wykonania programu. Nie mogą więc służyć do szacowania tego czasu. Ideą użycia miar uśrednionych jest uzyskanie informacji o stopniu wykorzystania sprzętu (stosunku miar uśrednionych do miar ekstremalnych) i przeprowadzanie na ich podstawie wnioskowania o możliwych wartościach miar dla innych programów lub dla innych egzemplarzy danych wejściowych, dla których nie dokonuje się pomiarów.

Uśredniona miara CPI (*average CPI*), dotycząca całości wykonania programu i będąca prostą odwrotnością miary IPC, definiowana jest jako

$$\mathrm{CPI} = \frac{1}{\mathrm{IPC}} = \frac{c^{ins}}{l^{ins}}$$

W takim ujęciu CPI oznacza średnią liczbę taktów zegara przypadającą na pojedynczy wykonywany rozkaz. W efekcie, przetwarzanie bez współbieżności charakteryzować się będzie wartościami CPI większymi niż jeden, natomiast idealne przetwarzanie potokowe dążyć będzie do wartości CPI równej jeden.

Powyższe definicje i analizy są podstawą tzw. równania wydajności (*performance equation*), realizującego ideę użycia miary CPI do szacowania czasu wykonania. Czas wykonania programu jest rozbijany na iloczyn trzech czynników:

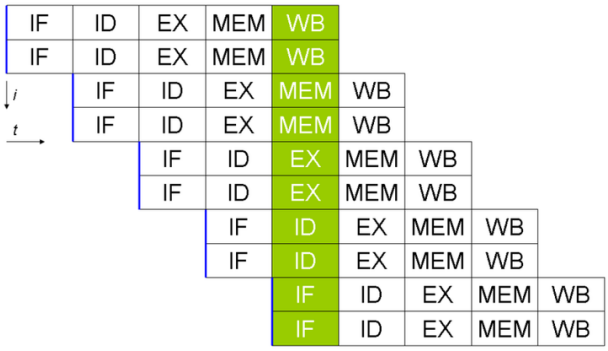
$$\text{czas wykonania} = \frac{\text{liczba sekund}}{\text{program}} = \frac{\text{liczba sekund}}{\text{takt}} \cdot \frac{\text{liczba taktów}}{\text{rozkaz}} \cdot \frac{\text{liczba rozkazów}}{\text{program}}$$

Ostatni czynnik zależy od kodu źródłowego i strategii doboru rozkazów procesora przez kompilator. Drugim czynnikiem jest parametr CPI dla konkretnego wykonania programu, a pierwszym czas trwania pojedynczego taktu procesora, będący odwrotnością częstotliwości jego pracy.

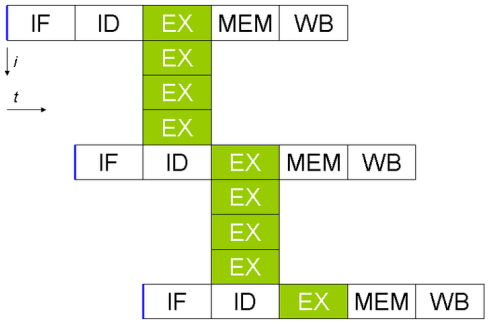
W praktyce uzyskanie wartości liczbowych każdego z powyższych czynników napotyka rozmaite trudności. Nawet dla tego samego kodu źródłowego różne kompilatory, w szczególności stosując różne opcje optymalizacji, produkują różne sekwencje rozkazów procesora. Współczynnik CPI, definiowany w sposób określony powyżej, uśrednia wartości, które są nie tylko różne dla różnych rozkazów, ale także, uwzględniając możliwe opóźnienia przetwarzania potokowego, mogą być różne dla tego samego rozkazu, w zależności od tego jakie rozkazy są przetwarzane bezpośrednio przed i bezpośrednio po nim. Wreszcie, częstotliwość pracy współczesnych procesorów nie jest wartością stałą lecz zmienia się w czasie wykonania programów, najczęściej stosownie do realizowanej strategii oszczędzania energii.

Niemniej równanie wydajności pozostaje istotną wskazówką optymalizacji, rozumianej jako dążenie do redukcji czasu wykonania. Aby ją osiągnąć należy:

- zmniejszać liczbę rozkazów w kodzie (lub używać bardziej wydajnych rozkazów – np. wektorowych)
- umożliwiać procesorom (rdzeniom) sprawne realizowanie przetwarzania potokowego (maksymalizacja rzeczywistego, uśrednionego IPC i minimalizacja CPI)
- zwiększać częstotliwość pracy procesora (rdzenia)



Rysunek 3.5: Schemat przetwarzania potokowego superskalarnego[źródło: Wikipedia]



Rysunek 3.6: Schemat przetwarzania potokowego SIMD [źródło: Wikipedia]

3.4.3 Superskalarność

Przypadek CPI=IPC=1 nie jest maksymalną wydajnością współczesnych procesorów. Zwielokrotnienie liczby tranzystorów umieszczanych w pojedynczym układzie scalonym, umożliwiło w latach 90-tych XX wieku budowanie mikroprocesorów superskalarnych o zwielokrotnionych jednostkach funkcjonalnych. Jeśli założymy, że podwojone są wszystkie układy uczestniczące w przetwarzaniu potokowym, otrzymujemy możliwość w pełni równoległego przetwarzania dwóch rozkazów, nazywanego dwudrożnym przetwarzaniem superskalarnym (*2-way superscalar processing*), zilustrowanego na rys. 3.5. W przykładzie tym, na każdym etapie przetwarzania potokowego znajdują się dwa rozkazy i w konsekwencji w każdym takcie zegara kończone są dwa rozkazy. Praktycznie stosowane procesory miały i mają możliwości superskalarnego przetwarzania ograniczone do kilku rozkazów. Drożność procesorów, liczba równolegle przetwarzanych rozkazów, zazwyczaj nie przekracza ośmiu, stąd w praktyce spotyka się procesory dwu-, cztero-, sześcio- czy ośmiodrożne (*2-, 4-, 6-, 8-way superscalar*). We współczesnych procesorach, o rozdzielonych podukładach pobierania i wstępnego przetwarzania rozkazów oraz różnych typach potoków wykonywania rozkazów, możliwości przetwarzania superskalarnego charakteryzuje się często bardziej szczegółowo podając liczbę potoków każdego typu i liczbę rozkazów wydawanych (przekazywanych do wykonania potokom) w pojedynczym takcie (*multiple-issue processors*).

W typowych procesorach, np. z rodziny *x86*, selekcji do wykonania superskalarnego dokonuje układ procesora, niezależnie od kompilatora i kodu źródłowego. Architektury, gdzie kompilator w pojedynczym rozbudowanym słowie przekazuje do wykonania równoległego kilka rozkazów (*very long instruction word, VLIW, architectures*), okazały się w praktyce mniej wydajne od standardowych architektur RISC.

Jeśli wprowadza się zwielokrotnienie liczby potoków wykonania dla pojedynczego zdekodowanego rozkazu, jak na rys. 3.6, przetwarzanie staje się typowym działaniem dla architektury SIMD (*single*

0	1	2	3	4	5	6	7	8	9	10	11
IF	ID	EX	MEM	WB							
	IF	*	*	*	ID	EX	MEM	WB			
		IF	*	*	*	ID	EX	MEM	WB		
			IF	*	*	*	ID	EX	MEM	WB	
				IF	*	*	*	ID	EX	MEM	WB

Rysunek 3.7: Przestój przetwarzania potokowego [źródło: Wikipedia]

instruction multiple data). Jeden pobrany i zdekodowany rozkaz dotyczy wielu egzemplarzy danych i wykonywany jest przez wiele jednostek wykonania (często łączonych w jeden potok). Obie modyfikacje, superskalarność i przetwarzanie SIMD (nazywane także przetwarzaniem wektorowym[3]), powodują, że teoretyczna możliwa do uzyskania przez procesor wartość CPI spada poniżej 1, natomiast wartość IPC sięga kilku lub nawet kilkudziesięciu (jak np. w przypadku procesorów graficznych).

3.4.4 Problemy przetwarzania potokowego

Wysoka teoretyczna wartość IPC współczesnych procesorów rzadko kiedy osiągana jest w praktyce. Jedną z przyczyn są zaburzenia idealnego przetwarzania potokowego. Rysunek 3.7 przedstawia sytuację przestoju przetwarzania potokowego (*pipeline stall*), w której procesor nie jest w stanie rozpocząć fazy ID drugiego rozkazu, do momentu zakończenia fazy WB rozkazu pierwszego.

Istnieje szereg przyczyn przestojów przetwarzania potokowego. Wynikają one z działań zmierzających do uniknięcia tzw. hazardów (*hazards*), czyli sytuacji, w których standardowe (niemodyfikowane i niewstrzymywane) przetwarzanie potokowe prowadzi do ryzyka wystąpienia błędów przetwarzania. Istnieje kilka typów hazardów:

- hazardy zasobów – kiedy dwa lub więcej rozkazów chce jednocześnie korzystać z tych samych zasobów procesora
- hazardy sterowania – kiedy w kodzie pojawia się instrukcja skoku (warunkowego lub bezwarunkowego)
- hazardy danych – kiedy poprawne wykonanie kolejnego rozkazu wymaga znajomości wyniku wcześniejszego rozkazu, co wyklucza współbieżność (taka definicja obejmuje tylko hazardy odczyt-po-zapisie, *read-after-write*, nie obejmuje możliwych do usunięcia przez odpowiednie transformacje kodu hazardów zapis-po-zapisie, *write-after-write*, i zapis-po-odczycie, *write-after-read*).

Unikanie hazardów i związanych z nimi opóźnień przetwarzania potokowego jest z jednej strony celem projektowania procesorów, a z drugiej techniką optymalizacji kodu. W tym ostatnim przypadku najistotniejsze jest unikanie hazardów danych, które na poziomie kodu źródłowego przejawiają się w postaci zależności danych (*data dependency*) pomiędzy instrukcjami kodu (np. rzeczywistej zależności

[3]Historycznie nazwa przetwarzania wektorowego i procesorów wektorowych odnosiła się do procesorów zawierających starannie zaprojektowane skalarne potoki fazy wykonania rozkazów zmiennoprzecinkowych, podczas gdy przetwarzanie SIMD charakteryzowało tzw. procesory macierzowe – w obu wypadkach wysoka przepustowość wykonania opierała się na realizacji pojedynczego rozkazu dla wielu egzemplarzy danych.

odczyt-po-zapisie, kiedy jedna instrukcja korzysta z wartości zmiennej uzyskanej w wyniku wykonania wcześniejszej instrukcji)[4]. Powiązanie zwiększania wydajności przetwarzania, poprzez podnoszenie stopnia współbieżności, z niezależnością instrukcji kodu źródłowego i rozkazów asemblera będzie tematem często pojawiającym się w dalszych analizach wydajności obliczeń.

Na poziomie projektowania procesorów istnieje szereg technik unikania hazardów, wprowadzających mniejsze lub większe opóźnienia przetwarzania, z których większością nie daje się sterować poprzez optymalizację kodu przez programistę. Dwie spośród technik najważniejszych dla wydajności przetwarzania, to przewidywanie rozgałęzień (skoków, *branch prediction*) i wykonywanie poza kolejnością (*out-of-order execution*).

Przewidywanie skoków (rozgałęzień)

Pierwsza z technik, przewidywanie rozgałęzień, ma na celu uniknięcie przestojów spowodowanych dążeniem do uniknięcia hazardów sterowania w przypadku skoków warunkowych. W przetwarzaniu niezmodyfikowanym, w sytuacji kiedy w potoku pojawia się rozkaz skoku warunkowego, informacja o tym, który rozkaz zostanie wykonany jako następny staje się dostępna dopiero po obliczeniu wartości logicznej warunku. W technice przewidywania rozgałęzień wykorzystuje się fakt, że większość rozgałęzień (skoków) w kodzie (np. związanych z wykonywaniem pętli) wykonywana jest wielokrotnie, a częstość spełniania lub nie warunku układa się w powtarzalny schemat (np. przy wykonywaniu długich pętli, skok na początek pętli realizowany jest miliony czy miliardy razy, natomiast przejście dalej tylko raz).

W przewidywaniu rozgałęzień, na podstawie wcześniejszych wartości logicznych warunku związanego ze skokiem, przewidywana jest wartość aktualna i realizowany skok związany z tą wartością. W przypadku kiedy wartość jest przewidziana prawidłowo eliminuje się przestój potoku. W przypadku błędnej predykcji, wykonanie cofa się ponownie do rozkazu skoku, a potok jest czyszczony (*flushed*) z efektów wszystkich rozkazów, które nastąpiły po nim. Technika ta, będąca przykładem wykonania spekulatywnego (*speculative execution*), powoduje w przypadku błędnego przewidywania skoków większe opóźnienie niż w przypadku wyłącznie wstrzymania przetwarzania niezmodyfikowanego, dlatego ostateczna przydatność przewidywania rozgałęzień zależy od jego skuteczności.

Istnieje szereg praktycznych realizacji mechanizmu przewidywania rozgałęzień. Potrafią one osiągać we współczesnych procesorach skuteczność, dla przeciętnych programów, rzędu 80-90 procent przypadków, a dla specyficznych fragmentów kodu (jak wspomniane wcześniej długie pętle) sięgać blisko 100 procent. Ma to istotne znaczenie dla wydajności, ponieważ w przeciętnych programach rozkazy skoku warunkowego (ze względu m.in. na powszechność stosowania pętli) mogą stanowić kilka-kilkanaście procent całkowitej liczby rozkazów wykonywanych w trakcie realizacji programu.

Wykonywanie poza kolejnością

Druga z ważnych technik unikania opóźnień przetwarzania potokowego, wykonywanie poza kolejnością, wymaga istotnych zmian w układach procesora. Pobierane rozkazy gromadzone są w odpowiednich strukturach danych, przeglądane, a następnie dokonywane jest ustalenie kolejności wykonania i przeprowadzane są ewentualne modyfikacje rozkazów (np. zmiana nazw rejestrów rozkazu). Kolejność wykonania rozkazów przez procesor jest ustalana wewnętrznie, jednak efekt zewnętrzny musi być identyczny jak w przypadku wykonania w kolejności wynikającej z zapisu kodu binarnego w pamięci (dotyczy to jednak tylko pojedynczego wątku, a więc pojedynczej sekwencji rozkazów). W związku z

[4]Zależność danych pomiędzy dwoma instrukcjami kodu powstaje, kiedy obie korzystają z tej samej komórki pamięci (tej samej zmiennej) i choć jedna z nich dokonuje zapisu – współbieżne wykonanie takich instrukcji prowadzi do wyścigu (*race condition*), niedeterministycznego wykonania programu, kiedy wynik zależy od kolejności wykonania instrukcji, która nie jest poddana kontroli.

tym istnieje jeszcze jeden etap, porządkujący efekty pracy procesora, etap opuszczania procesora przez rozkaz (*instruction retirement*).

Układy wykonywania poza kolejnością zabierają znaczną liczbę tranzystorów i powierzchni procesora, co np. oznacza mniejszą liczbę tranzystorów na układy wykonania rozkazów, jednak ich działanie jest na tyle ważne, że stanowią element wszystkich procesorów ogólnego przeznaczenia (o specjalnych procesorach, w których rozkazy wykonywane są w kolejności, mowa będzie w dalszej części książki).

Ostatnią omawianą, choć w praktyce zazwyczaj najważniejszą, przyczyną opóźnień przetwarzania potokowego jest przyczyna leżąca poza układem wykonania rozkazów przez procesor. Wstrzymanie przetwarzania ma miejsce w sytuacji, kiedy nie zostały dostarczone dane potrzebne do wykonania rozkazu. Częściowo problem może zostać rozwiązany przez wykorzystanie wykonania poza kolejnością (lub wielowątkowość, o czym będzie mowa w kolejnych punktach), jednak podstawowe znaczenie ma optymalizacja układu dostarczania danych, związana z organizacją pamięci w systemie komputerowym.

3.5 Wielopoziomowa organizacja pamięci

W klasycznej architekturze von Neumanna, dla której wciąż jeszcze tworzony jest w przypadku większości procesorów kod binarny (z możliwym zapisem w językach asemblera), istnieją tylko dwa typy pamięci jako argumenty rozkazów: rejestry i pamięć główna.

Rejestry stanowiące wewnętrzną pamięć procesora, w szczególności ich liczba i typy (określające najczęściej rozmiar w bitach oraz przeznaczenie), decydują w istotnej mierze o możliwościach wydajnościowych procesora. Możliwość jawnego wykorzystania rejestrów w programowaniu istnieje tylko w przypadku stosowania asemblera lub specjalnych rozszerzeń języków programowania, występujących często w postaci zbliżonych do asemblera wstawek (*intrinsics*) rozpoznawanych przez kompilatory języków wyższego poziomu.

W języku asemblera dostęp do pamięci głównej odbywa się poprzez użycie rozkazów procesora, w których jako argument występuje adres pamięci, przechowywany w rejestrze lub obliczany na podstawie wartości przechowywanych w kilku rejestrach, dla złożonych trybów adresowania (p. 3.2). Rozmiar używanych w tym celu rejestrów (8, 16, 32, 64 bity) jest związany z rozmiarem dostępnej przestrzeni adresowej (2^n komórek pamięci, gdzie n jest liczbą bitów rejestrów), wpływa na konstrukcję i funkcjonowanie układów pamięci oraz magistral łączących procesor z pamięcią.

Jedną z podstawowych, ze względu na wydajność, modyfikacji standardowego modelu pamięci architektury von Neumanna, jest wprowadzenie pamięci podręcznej (*cache memory*). Jej całościowemu omówieniu poświęcony jest jeden z kolejnych podrozdziałów książki. W tym miejscu zaznaczony jest tylko jej podstawowy schemat: zamiast jednego poziomu pamięci (a więc sytuacji, kiedy wartość konkretnej zmiennej przechowywana jest zawsze w jednej tylko lokalizacji), istnieje wiele poziomów, tak że z jedną zmienną w programie może być związane kilka lokalizacji (w pamięci głównej i różnych poziomach pamięci podręcznej), z wartością najbardziej aktualną przechowywaną często tylko w niektórych lokalizacjach.

Najbliżej potoków procesora (pomijając rejestry) znajduje się poziom L1 pamięci podręcznej. W najpopularniejszych architekturach procesorów, poziom ten zorganizowany jest w sposób odpowiadający tzw. architekturze harvardzkiej. Pamięć główna (DRAM), zgodnie z architekturą von Neumanna, przechowuje w jednej przestrzeni adresowej kod programu i jego dane, w architekturze harvardzkiej istnieją osobne pamięci dla kodu i dla danych.

We współczesnych procesorach pobranie lub zapis dowolnego argumentu rozkazu asemblera z lub do pamięci głównej jest złożonym procesem. Adres zapisany w rejestrze nie jest adresem fizycznym wykorzystywanym przy sprzętowym dostępie do układów pamięci i musi zostać na taki adres przetłumaczony. Stosuje się w tym celu rozmaite techniki, zależne od systemu operacyjnego, z których najpopularniejszą

dzisiaj jest technika pamięci wirtualnej (omówiona szerzej w p. 4.1). Architektury procesorów wspierają stosowanie pamięci wirtualnej, poprzez przechowywanie związanej z nią tablicy stron, wspomagającej tłumaczenie adresów z wirtualnych na fizyczne, w pełnej hierarchii pamięci. Tablica stron posiada więc specjalne dedykowane pamięci podręczne, zwane także buforami translacji adresów (TLB, *Translation Lookaside Buffer*), w tym pamięć TLB poziomu L1 bezpośrednio w procesorze.

3.6 Przetwarzanie SIMD i wektoryzacja

Wspomniane w punkcie 3.4.3, omawiającym modyfikacje klasycznego przetwarzania potokowego, przetwarzanie SIMD jest na poziomie procesora związane z wykorzystaniem rejestrów wektorowych. Rejestr wektorowy to odpowiednio szeroki rejestr (w klasycznej architekturze x86 i jej rozszerzeniach mający od 64 do 512 bitów), który może być użyty do przechowania kilku egzemplarzy liczb określonego typu. Przykładowo, rejestr 128-bitowy może pomieścić cztery liczby 32-bitowe, całkowite lub zmiennoprzecinkowe pojedynczej precyzji, 2 liczby 64-bitowe lub większą liczbę zmiennych o typach wymagających 8 lub 16 bitów.

Z rejestrami wektorowymi określonego typu (czyli określonej szerokości) związany jest specjalny zestaw rozkazów na liście rozkazów procesora. Współczesne procesory udostępniają po kilka zestawów rejestrów (różne procesory oferują różną liczbę i różny zakres szerokości rejestrów), często opatrując zestawy rozkazów operujących na danych rejestrach specjalnymi nazwami (np. MMX, 3DNOW, SSE). Odwołania w tych nazwach do pojęć związanych z renderowaniem grafiki (np. MMX – *Multimedia Extensions*), wynikają z pierwotnego ukierunkowania wykorzystania rozkazów wektorowych wyłącznie na przetwarzanie grafiki. Jednak dzisiaj, w sytuacji kiedy rejestry wektorowe osiągają rozmiar 512 bitów, każda dziedzina zastosowań powinna starać się maksymalizować wykorzystanie rejestrów wektorowych. W podanym przykładzie rejestrów 512-bitowych, wykorzystanie jednego rozkazu wektorowego dla liczb całkowitych jest równoważne realizacji 16 odpowiadających operacji skalarnych. Odpowiednia organizacja obliczeń pozwala ukryć narzut związany z wykonaniem wektorowym, wynikający m.in. z konieczności pakowania i rozpakowywania kilku czy kilkunastu egzemplarzy danych do i z rejestrów wektorowych, i uzyskać przyspieszenie obliczeń wprost związane z szerokościa rejestrów, czyli w rozważanym przypadku przyspieszenie 16-krotne.

Patrząc na ten aspekt z innej strony, można uznać, że nie zastosowanie rozkazów wektorowych oznacza kilku- lub kilkunastokrotne zmniejszenie wydajności w stosunku do maksymalnej oferowanej przez procesor.

Z przetwarzaniem SIMD wiąże się jeszcze jedno pojęcie wygodne w analizie wydajności. Rozkazy wektorowe są wykonywane na argumentach będących rejestrami wektorowymi, przy wykorzystaniu specjalnych wektorowych potoków przetwarzania. Działanie takiego potoku można sobie wyobrazić jako działanie, w pełnej synchronizacji sprzętowej, kilku pojedynczych potoków przetwarzania skalarnego. Tak wyodrębniony potok odpowiadający przetwarzaniu skalarnemu będzie dalej nazywany ścieżką SIMD (*SIMD lane*).

W prezentowanym ujęciu, wykonanie pojedynczego rozkazu wektorowego jest związane z wykorzystaniem tylu ścieżek SIMD, ile egzemplarzy danych jest spakowane w pojedynczym rejestrze wektorowym, natomiast wykonanie dowolnej operacji skalarnej jest związane z pojedynczą ścieżką SIMD. Występowanie na schemacie procesora potoku przetwarzania skalarnego określonego typu oznacza istnienie w procesorze pojedynczej ścieżki SIMD tego typu. Występowanie potoku przetwarzania wektorowego oznacza istnienie kilku ścieżek SIMD, jednak ich liczba zależy od typu rejestrów i zestawu powiązanych rozkazów. Potok przetwarzania rozkazów 256-bitowych jest równoważny istnieniu 8 ścieżek SIMD dla zmiennych 32-bitowych i 4 ścieżek dla zmiennych 64-bitowych. Jest to prawdziwe w sytuacji kiedy te same potoki wektorowe obsługują różne typy danych. W wielu architekturach występują odrębne potoki

przetwarzania dla różnych typów danych, w szczególności 32 i 64 bitowych. W takim przypadku liczba ścieżek SIMD jest odrębnie określana dla każdego z typów danych.

Ścieżki SIMD są wygodnym narzędziem charakteryzowania maksymalnej wydajności przetwarzania dla rozkazów konkretnego typu. Maksymalna wydajność procesora to łączna wydajność wszystkich ścieżek SIMD danego typu pracujących jednocześnie. Liczba ścieżek SIMD to liczba jednocześnie wykonywanych operacji skalarnych danego typu. Często występującym przypadkiem jest sytuacja kiedy maksymalna wydajność osiągana jest w wyniku pracy kilku potoków wektorowych. Wtedy wydajność procesora dla danego typu operacji jest iloczynem liczby potoków i liczby egzemplarzy danych w jednym rejestrze wektorowym danego typu (równa liczbie ścieżek SIMD w potoku). Wydajność taka określa maksymalną liczbę skalarnych operacji danego typu (np. zmiennoprzecinkowych pojedynczej lub podwójnej precyzji) kończonych w pojedynczym takcie procesora (zakładając, że potoki pozwalają na kończenie jednego rozkazu wektorowego w każdym takcie). Po pomnożeniu przez częstotliwość pracy procesora uzyskujemy maksymalną wydajność procesora w liczbie operacji na sekundę.

W dalszej części książki, jako sprzęt na którym uruchamiane są przykładowe programy, wykorzystywane jest kilka mikroprocesorów. Jednym z nich jest 4-rdzeniowy mikroprocesor Intel Core i7-4790, z rdzeniami o architekturze Haswell i nominalną częstotliwością pracy 3.60 GHz. Jego każdy rdzeń posiada dwa potoki przetwarzania rozkazów 256-bitowych. Każdy z potoków potrafi w jednym takcie kończyć jedną połączoną operację mnożenia i dodawania (*fused multiply-add – FMA*). Dla zmiennych podwójnej precyzji oznacza to 8 (4·2) skalarnych operacji kończonych w każdym takcie. Ostateczna maksymalna wydajność pojedynczego rdzenia wynosi $8 \cdot 2 \cdot 3.6 \cdot 10^9 = 57.6 \cdot 10^9$ operacji arytmetycznych podwójnej precyzji na sekundę, czyli 57.6 Gflop/s (*floating point operations per second*), co prowadzi do maksymalnej wydajności czterech rdzeni mikroprocesora 230.4 Gflop/s.

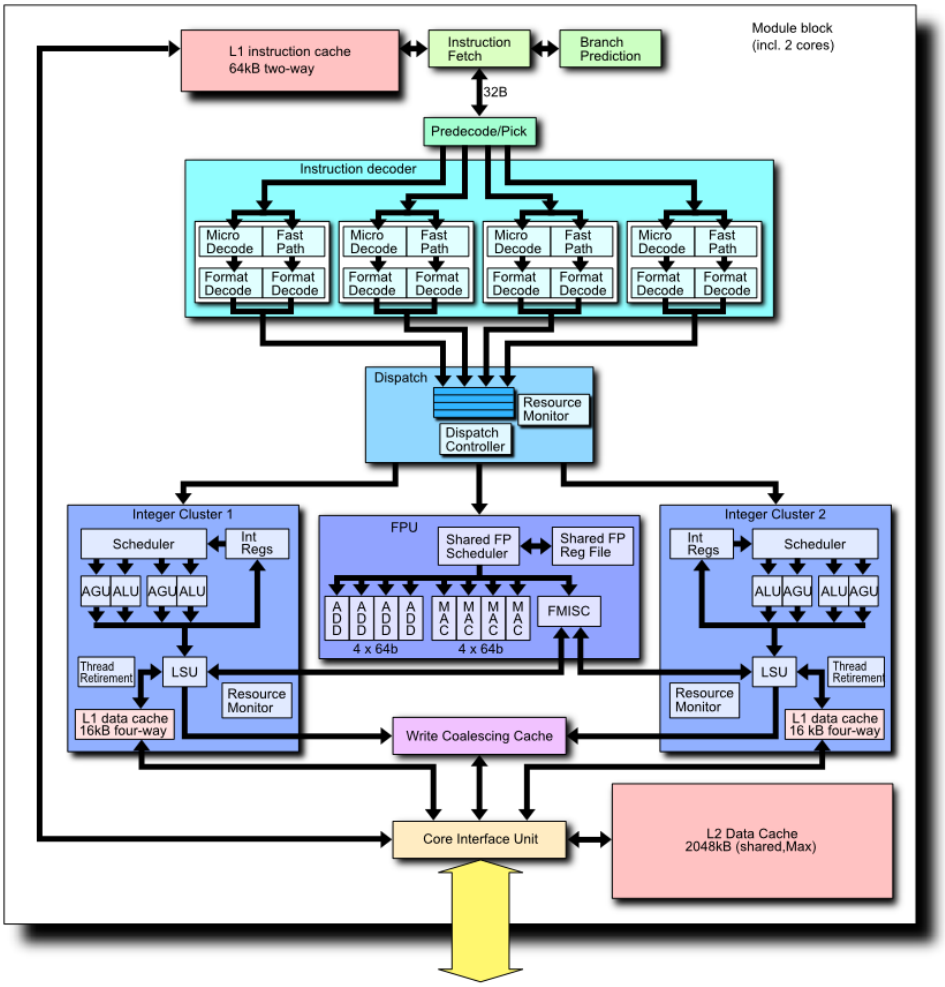
3.7 Przykłady mikroarchitektur procesorów

Rysunki 3.8, 3.9 i 3.10 przedstawiają przykłady diagramów blokowych pojedynczych rdzeni współczesnych mikroprocesorów, jako odpowiedników klasycznej maszyny von Neumanna.

Większość współczesnych procesorów (rdzeni), w tym przykładowe (mikro-)architektury AMD Bulldozer, ARMv8 i Intel Core 2 (a także nowsze), posiada zbliżoną do siebie budowę. Można w nich wyróżnić dwie podstawowe grupy układów: system pobierania i dekodowania rozkazów oraz system wykonywania rozkazów, a także występujący pomiędzy nimi układ planowania (*schedule*) i rozdysponowania (*dispatch*) rozkazów. Poniżej omówione są podstawowe elementy przykładowych architektur (stosując wspólne nazwy układów, a w przypadku gdy oznaczenia układów na diagramach są różne, używając w tekście kolejności odpowiadającej kolejności rysunków), z pominięciem niektórych bloków oraz bardziej szczegółowych informacji zawartych na diagramach .

Analizując budowę procesorów w kolejności odpowiadającej kolejności przetwarzania pojedynczego rozkazu, jako pierwsze występują układy pobierania rozkazów. Źródłem rozkazów jest pamięć podręczna L1 rozkazów, oznaczana na diagramach jako blok *(L1) Instruction Cache*. Rozkazy z pamięci podręcznej, mającej typowe rozmiary 32, 64 kB, trafiają do układu pobierania, który przechowuje jednocześnie wiele rozkazów w odpowiednich strukturach danych (bloki: AMD – *Instruction Fetch*, ARM – *Fetch Queue*, Intel – *Fetch Buffer, Instruction Queue, Instruction Fetch Unit*). Pobieranie rozkazów wymaga translacji ich adresów, do czego służą układy pamieci TLB, występujacej także w wariancie dedykowanym wyłącznie dla rozkazów (Intel – *ITLB*).

Umieszczenie zbioru rozkazów w strukturach danych procesora umożliwia ich wstępne zdekodowanie, przeglądnięcie zawartości i operacje takie jak przewidywanie rozgałęzień (bloki *Branch Prediction*, w architekturze Intel wewnątrz *Instruction Fetch Unit*).



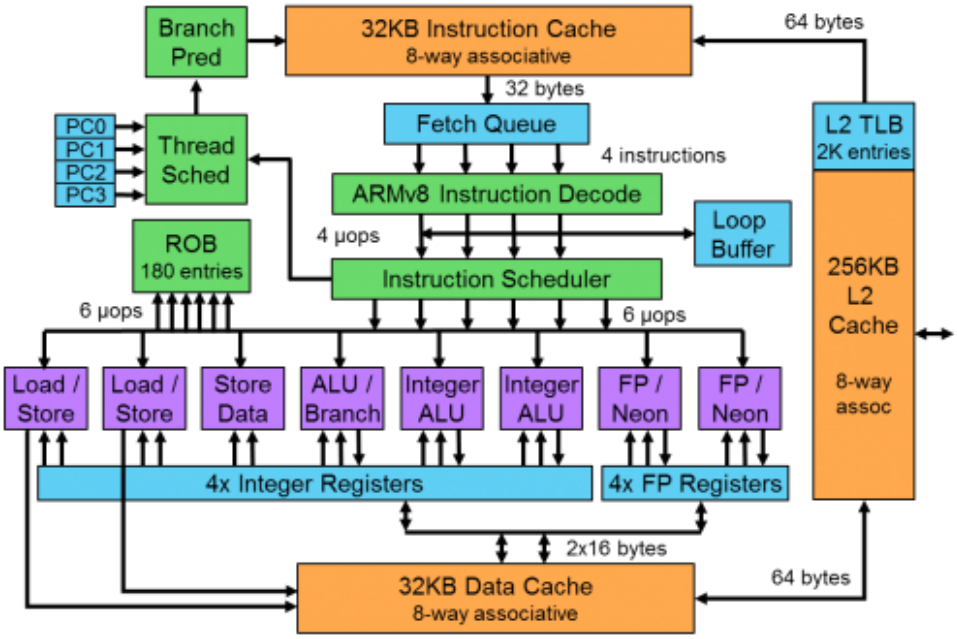
Rysunek 3.8: Architektura AMD Bulldozer [źródło: Wikipedia]

Dalszym etapem przetwarzania jest pełne dekodowanie rozkazów, do postaci właściwej dla rozdysponowania (wydania) do poszczególnych potoków wykonania. Wykonanie rozkazów odbywa się przez potoki specjalnie zaprojektowane do realizacji różnych typów rozkazów. Każdy rozkaz określonego typu rozbijany jest na właściwą sobie liczbę faz i wykonywany przy użyciu odpowiednich zasobów procesora. Na przykładowych diagramach znajdują się potoki:

- ALU (*Arithmetic-Logic Unit*), *Integer ALU* – klasyczne potoki realizacji operacji arytmetyczno-logicznych, wykonywanych na liczbach całkowitych

- AGU (*Address Generation Unit*), LSU (*Load/Store Unit*), *Load/Store Adress*, *Load/Store Data* – jednostki pobierania danych oraz rozkazów z hierarchii pamięci i zapisywania do pamięci, w tym ewentualnie odrębne jednostki tłumaczenia i obliczania (generowania) adresu (związane m.in. z faktem stosowania pamięci wirtualnej i występowania w asemblerze rozkazów ze złożonymi trybami adresowania)

- FP, (F)ADD, (F)MUL, (F)MAC, F(MAC), (F)DIV – jednostki wykonywania operacji na argumentach zmiennoprzecinkowych (dodawanie, mnożenie, dzielenie, łączne dodawanie i mnożenie



Rysunek 3.9: Architektura ARMv8 [źródło: Wikipedia]

– oznaczane jako MAC – *Multiply-Accumulate*, MAD – *Multiply-Add* lub FMA – *Fused Multiply-Add*), obejmujące także jednostki wykonywania rozkazów SIMD (o nazwach specyficznych dla rodziny procesorów, np. SSE, Neon, AVX, VMX)

- *Branch*, *Shuffle* – jednostki wykonywania specjalnych typów rozkazów

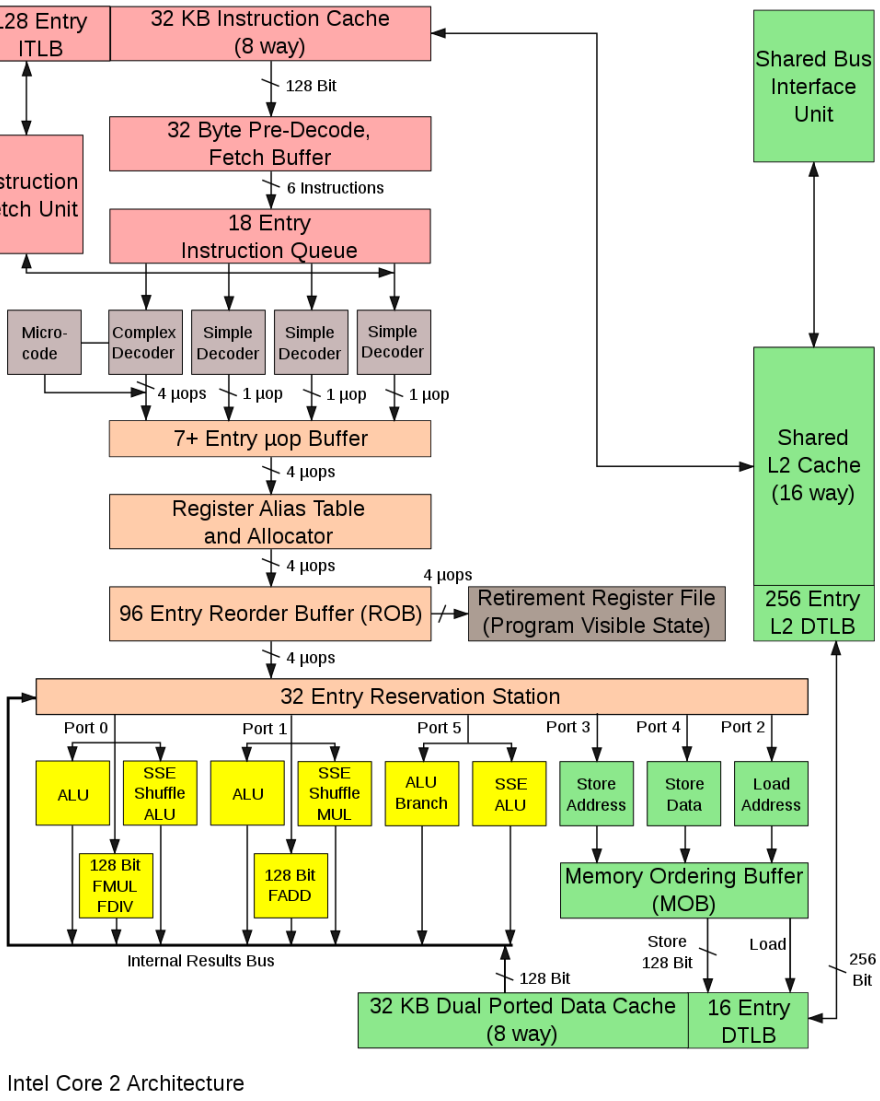
Występowanie wielu, w praktyce niezależnych, potoków wymusza istnienie rozbudowanych układów pomiędzy podsystemem pobierania i dekodowania rozkazów, a jednostkami (potokami) wykonywania rozkazów (stosuje się także pojęcie portów, jawnie zaznaczonych na diagramie mikroarchitektury firmy Intel, jako jednostek rozdysponowania rozkazów). Organizacja i nadzorowanie pracy jednostek przetwarzania potokowego wymagają szeregu działań, takich jak: rozdzielanie rozkazów pomiędzy potoki, zarządzanie przydziałem rejestrów dla rozkazów (co związane może być z przemianowaniem rejestrów, *register renaming*), porządkowanie opuszczania procesora przez rozkazy, *instruction retirement*. Działania te realizowane są przez układy oznaczane na diagramach jako *Dispatcher* (dyspozytor), *Scheduler* (planista) i wykorzystują dodatkowe, zaznaczone na diagramach podukłady i struktury danych (jak np. ROB, *Reorder Buffer* lub *Retirement Register File*).

3.7.1 Optymalizacja przetwarzania przez pojedynczy rdzeń mikroprocesora

Jak widać na przykładach zaprezentowanych mikroarchitektur, budowa pojedynczego rdzenia mikroprocesora jest na tyle złożona, że optymalizacja jego pracy przy realizacji konkretnego programu wymaga uwzględnienia szeregu aspektów, takich jak między innymi:

- liczba rozkazów w kodzie asemblera – im wyższa tym większe wymagania szybkości pobierania i dekodowania rozkazów przez procesor, w szczególności w przypadku rozkazów o krótkim czasie wykonania

- złożoność rozkazów, w tym złożoność trybów adresowania – wpływające na wymagania szybkości dekodowania, a także na liczbę pojedynczych operacji związanych z realizacją rozkazów



Rysunek 3.10: Architektura Intel Core 2 [źródło: Wikipedia]

(pośrednio na liczbę potoków przetwarzania zaangażowanych w realizację rozkazu – zazwyczaj potoki przetwarzania związane są z mikro-rozkazami na które rozbijane są rozkazy z listy rozkazów procesora)

- sposób organizacji obliczeń – powiązany z możliwościami efektywnego użycia układów przewidywania rozgałęzień i wykonywania poza kolejnością
- wzajemne zależności między rozkazami – określające hazardy przetwarzania, a więc i opóźnienia przetwarzania potokowego
- organizacja przechowywania struktur danych oraz sposoby korzystania z pamięci głównej

Ostatni z wymienionych aspektów w przypadku wielu algorytmów okazuje się być decydującym o ostatecznej wydajności. Jest jednakże jednym z aspektów najbardziej złożonych – obejmuje nie tylko liczbę dostępów do danych, ale także organizację tych dostępów, ich umiejscowienie w kodzie asemblera i wzajemne uporządkowanie, co w dalszej kolejności określa korzystanie z pamięci podręcznej

rozmaitych poziomów, a także pamięci wirtualnej z obsługą tablicy stron i związanych z nią pamięci podręcznych ITLB i DTLB .

Z punktu widzenia wydajności idealną sytuacją jest pełne wykorzystanie wszystkich jednostek potokowych procesora, kiedy wszystkie jednostki pracują równolegle, a ich potoki nie mają przestojów. Układ pobierania i dekodowania rozkazów oraz układ rozdysponowania rozkazów są w stanie dostarczać w każdym takcie każdemu potokowi rozkazy do wykonania wraz z potrzebnymi danymi. Taka wizja pracy procesora prowadzi do modelu analizy wydajności, opierającego się na założeniu, że w przypadku, gdy wydajność odbiega od optymalnej, badane jest, które elementy okazały się wydajnościowym wąskim gardłem (*performance bottleneck*), decydującym o przestojach w pracy pozostałych układów i wydłużeniu czasu realizacji obliczeń.

Powyższy model analizy ma swoje wady. Bada wykorzystanie procesora jako całości, bez względu na specyfikę wykonywanego programu. Czasem postać kodu źródłowego jawnie determinuje, które elementy procesora będą wykorzystane, a które nie. Na przykład, w przypadku kodu o zdecydowanej przewadze operacji zmiennoprzecinkowych, potoki przetwarzania liczb całkowitych będą mało wykorzystane, a uśrednione wyniki wykorzystania wszystkich potoków mało reprezentatywne dla rzeczywistej wydajności przetwarzania.

Z tych względów, w niniejszej książce przyjęty jest inny kierunek badania. Punktem wyjścia są algorytmy, a nie poszczególne układy procesora. Istotne (przynajmniej dla określonych dziedzin zastosowań) algorytmy i ich implementacje są analizowane, określane są czynniki decydujące o ich wydajności, tworzone modele wydajności i poszukiwane efektywne metody optymalizacji. Dla każdego programu rozważane są tylko potoki przetwarzania istotne dla realizacji tego właśnie programu, w konkretnym momencie jego wykonania.

Specyfika analizowanych w książce algorytmów sprawia, że przy analizie ich wydajności nie pojawia się problematyka pobierania i dekodowania rozkazów – w przypadku rozważanych programów etapy te nie stanowią czynnika ograniczającego wydajność dla współczesnych procesorów.

3.8 Prawo Little'a

Często przywoływanym w kontekście wydajności układów przetwarzania potokowego, a także innych układów przetwarzania współbieżnego, jest prawo Little'a, oryginalnie sformułowane w teorii kolejek.

W pierwotnym ujęciu, rozpatrywanymi elementami jest napływający strumień klientów oraz układ wykonywania ich żądań. Przybywający klienci wchodzą do systemu, a po obsłużeniu opuszczają system.

Prawo Little'a stwierdza, że w przypadku systemów stacjonarnych uśredniona liczba klientów obsługiwanych wewnątrz systemu (L) jest równa iloczynowi uśrednionego tempa przybywania klientów (λ) i średniego czasu przebywania w systemie (W):

$$L = \lambda \cdot W$$

(w powyższym wzorze zastosowane są klasyczne oznaczenia z prawa Little'a, różne od standardowych oznaczeń w pozostałej części książki).

W badaniu wydajności rdzeni mikroprocesora rolę strumienia klientów pełni strumień rozkazów (instrukcji do przetworzenia), a układem wykonywania żądań jest układ realizacji rozkazów (instrukcji), czyli potoki przetwarzania. Optymalizacja przetwarzania oznacza pełne wykorzystanie możliwości sprzętu, a więc całkowite zapełnienie wszystkich potoków przetwarzania.

Wymaganie stacjonarności systemu będzie w takim przypadku oznaczać, że uśrednione liczby rozkazów przybywających (przekazywanych do realizacji) oraz opuszczających system są sobie równe, a w konsekwencji parametr λ określa przepustowość układu. Miarą takiej przepustowości może być wartość

IPC lub liczba rozkazów kończonych w jednostce czasu (wyrażana jako iloczyn IPC i częstotliwości pracy rdzenia ν).

W przypadku potokowego przetwarzania rozkazów, czas przebywania w systemie (W) to liczba etapów przetwarzania potokowego pojedynczego rozkazu (k), wyrażana najczęściej w taktach (zakładając jeden etap na takt) lub jednostkach czasu, po pomnożeniu liczby taktów przez czas pojedynczego taktu, będący odwrotnością częstotliwości pracy procesora ν.

W efekcie tych założeń liczba L rozkazów jednocześnie (współbieżnie) przetwarzanych przez układ wynosi:

$$L = \lambda \cdot W = IPC \cdot \nu \cdot \frac{k}{\nu} = IPC \cdot k$$

Tak sformułowane prawo można odczytywać jako powiązanie opóźnienia i przepustowości przetwarzania rozkazów przez układ w postaci jednego lub wielu potoków oraz wymaganego stopnia współbieżności przetwarzania.

Kluczowym w analizie wydajności wnioskiem z prawa Little'a jest, że jeśli chcemy maksymalizować przepustowość przetwarzania przez potoki (IPC), zakładając, że liczba etapów potoku k jest z góry zadaną wartością, musimy maksymalizować liczbę współbieżnie przetwarzanych rozkazów (L). Oznacza to, że aby w pełni wykorzystać możliwości (maksymalną teoretyczną przepustowość) sprzętu musimy posiadać w kodzie odpowiednio dużą liczbę rozkazów do przetwarzania współbieżnego, a więc rozkazów niezależnych. Zapewnienie w kodzie źródłowym odpowiednio dużej liczby niezależnych od siebie instrukcji (m.in. niepowiązanych ze sobą zależnościami danych) staje się jednym z podstawowych wymagań dla pełnego wykorzystania możliwości pojedynczego rdzenia mikroprocesora.

3.9 Liczniki sprzętowe

Cennym narzędziem wspomagającym badania funkcjonowania mikroprocesorów, oraz współpracujących z nimi innych układów, są liczniki sprzętowe (*hardware counters*), wspomniane już w p. 2.2. Idea liczników sprzętowych polega na wyodrębnieniu szczegółowych zdarzeń związanych z realizacją programów i zliczaniu ich wystąpień, najczęściej w trakcie wykonania konkretnego programu, a czasem, w określonym przedziale czasu, dla całego systemu lub wybranego układu (np. pojedynczego rdzenia mikroprocesora). Najważniejszym odnotowywanym zdarzeniem jest zawsze pojedynczy takt zegara – dzięki temu można określić liczbę taktów w trakcie wykonania programu. Liczba taktów stosowana jest także do uzyskania pochodnych miar wydajności, wyrażanych w liczbie zdarzeń przypadających na pojedynczy takt (lub liczbie taktów na pojedyncze analizowane zdarzenie).

Mechanizm sprzętowy wykorzystywany w licznikach to specjalne dedykowane rejestry oraz odpowiednie układy zwiększania wartości rejestrów w przypadku zajścia konkretnego zdarzenia. Dla każdego rdzenia mikroprocesora istnieje tylko kilka rejestrów do zliczania zdarzeń, co oznacza, że dla każdego wykonania programu można śledzić częstość występowania tylko kilku zdarzeń. Jednak rejestry w większości są rejestrami ogólnego przeznaczenia (z wyjątkiem paru, np. rejestru na stałe przypisanego do zliczania taktów zegara) i przy każdym uruchomieniu programu można zlecać zliczanie innych zdarzeń. Istnieje kilka możliwych sposobów dostępu do rejestrów zliczających zdarzenia (ich resetowania, przypisywania do konkretnych zdarzeń, odczytywania wartości), tak z poziomu rozkazów asemblera, funkcji systemowych, jak i odpowiednich programów narzędziowych.

Początkowo liczniki sprzętowe obejmowały podstawowe zdarzenia, jak np. wykonywane operacje arytmetyczne z podziałem na całkowite i zmiennoprzecinkowe, skoki warunkowe i bezwarunkowe itp. Z czasem liczba zdarzeń możliwych do zliczania dla konkretnego procesora/rdzenia rosła, osiągając dziś kilkaset i obejmując także zdarzenia zachodzące poza rdzeniem. Często zdarzenia zdefiniowane dla procesora są bardzo szczegółowe (np. "liczba taktów w trakcie gdy mikrooperacje inicjowane przez bufor

dekodowania są dostarczane do kolejki dekodowania rozkazów, podczas gdy układ kolejkowania rozkazów jest zajęty"), co związane jest z faktem, że podstawową rolą liczników sprzętowych jest wspieranie procesu analizowania i projektowania budowy procesorów/rdzeni oraz ich pojedynczych detali architektonicznych.

Z punktu widzenia analizy wykonania programów, użycie liczników sprzętowych może napotykać trudności. Pewne zdarzenia, których zliczanie byłoby istotne dla analizy wydajności, czasem nie są definiowane (np. dla kilku generacji procesorów firmy Intel nie jest zdefiniowane zdarzenie wykonania operacji zmiennoprzecinkowej). Często interesujące zdarzenia są uwikłane w złożone mechanizmy działania procesora/rdzenia, przez co relatywnie prostym instrukcjom w programie mogą odpowiadać różne zdarzenia lub kombinacja kilku zdarzeń (dzieje się tak np. kiedy procesor wykonuje działania niewynikające bezpośrednio z kodu, jak w przypadku realizacji ścieżki wybranej przez układ przewidywania rozgałęzień, która w rzeczywistości nie jest wybrana, lub pobierania danych, które nie są później wykorzystywane w programie[5]).

W dalszych badaniach w książce wyniki zliczania zdarzeń sprzętowych, jako konkretne wartości, są uznawane za związane z konkretnym sprzętem, konkretnym wykonaniem i specyficznym dla tej sytuacji funkcjonowaniem mechanizmów zliczania, a przez to traktowane w specjalny sposób. Za podstawę badania przyjmowane są wartości z analiz teoretycznych, a wyniki zliczania zdarzeń, jeśli różnią się od przewidywanych teoretycznie, przyjmowane jako ewentualne, czasami trudne lub niemożliwe do precyzyjnego zinterpretowania, odchylenia od wartości podstawowych. Trudność interpretacji jest często związana z faktem, że uzyskane wyniki zliczania zdarzeń różnię się, i to w sposób istotny, dla różnych przypadków uruchomienia tego samego kodu. Niemniej zwracane przez narzędzia profilowania liczby zdarzeń pozostają istotna wskazówką, tego co w rzeczywistości dzieje się w układach mikroprocesora, i w przypadku, kiedy znacząco różnią się od wyników analiz teoretycznych, wymagają zbadania, co może prowadzić do korekty modelu teoretycznego.

3.9.1 Narzędzia umożliwiające dostęp do liczników sprzętowych

Pierwszym ze sposobów dostępu do rejestrów zliczających konkretne zdarzenia jest wykorzystanie w programach wstawek kodu asemblera zapisujących i odczytujących wartości odpowiednich rejestrów. Rozwiązanie to, poza brakiem przenośności między architekturami procesorów, może także wymagać wyższych uprawnień przy wykonywaniu kodu, niż dostępne standardowemu użytkownikowi. Z powodu tych wad wprowadzane są narzędzia wyższego poziomu. W systemie Linux istnieje narzędzie *perf*, korzystające z modułu wbudowanego w jądro Linuxa i służące do wspomagania analizy wydajności programów. Pozwala ono między innymi na uzyskiwanie wartości liczników sprzętowych dla zbioru wstępnie zdefiniowanych zdarzeń lub zdarzeń zadanych przez użytkownika, związanych z wykonywanym programem. W książce wykorzystywana będzie komenda *perf stat*, zwracająca za pomocą prostego interfejsu wartości liczników sprzętowych dla kilku wybranych, podstawowych zdarzeń w trakcie wykonania programu określonego jako argument komendy.

Do uzyskania wartości liczników sprzętowych dla fragmentów kodu używane będzie w książce narzędzie PAPI (*Performance Application Programming Interface*). Jego kolejne wersje dostarczają przenośny interfejs oraz realizującą go bibliotekę, umożliwiające dostęp do liczników sprzętowych dla kolejnych generacji mikroprocesorów, włączając w to także procesory graficzne. PAPI udostępnia wartości liczników dla wszystkich możliwych do zaprogramowania zdarzeń dla konkretnych procesorów, posługując się także zestawem wybranych, podstawowych zdarzeń (tzw. predefiniowane zdarzenia PAPI), traktowanych jako powszechnie występujące we wszystkich typach architektur.

[5] W ramach mechanizmu pobierania z wyprzedzeniem omawianego w dalszej części książki

3.9.2 Wykorzystanie liczników sprzętowych do określania częstotliwości pracy rdzenia

W pomiarach wydajności tradycyjnie używanymi miarami są najczęściej liczby konkretnych operacji (np. dostępów do pamięci lub operacji zmiennoprzecinkowych) wykonanych w jednostce czasu. Z perspektywy architektury procesora bardziej fundamentalnymi wartościami są liczby przypadające na pojedynczy takt jego pracy. Ze względu na zmienność częstotliwości pracy rdzeni we współczesnych mikroprocesorach te dwie grupy miar mogą różnić się dla różnych uruchomień programu. Podczas gdy miary liczby operacji na takt są zazwyczaj stałe (zwłaszcza w przypadku operacji bezpośrednio związanych z pracą rdzenia), liczby operacji w jednostce czasu będą wyższe przy pracy z wyższą częstotliwością, a niższe dla niższych częstotliwości. Jeśli rdzeń podczas wykonania programu pracuje z wyższą częstotliwością niż jego częstotliwość nominalna, mogą pojawić się wydajności wyższe niż teoretyczne maksima obliczone na podstawie nominalnych parametrów sprzętu.

W celu ujednolicenia wyników uzyskiwanych dla różnych uruchomień programu niezbędne staje się uwzględnienie rzeczywistej częstotliwości pracy rdzenia podczas wykonania programu. Liczniki sprzętowe są narzędziem, które umożliwia uzyskanie przybliżenia średniej częstotliwości pracy rdzenia przy wykonaniu nie tylko całego programu, ale także określonych jego fragmentów.

Aby uzyskać takie oszacowanie wykorzystywane jest zliczanie dwóch zdarzeń sprzętowych. Jednym z nich jest rzeczywisty takt zegara procesora (rdzenia) w trakcie wykonania kodu, drugim takt (referencyjny) zegara odmierzającego czas zgodnie z nominalną częstotliwością pracy procesora (rdzenia). Liczba taktów pomnożona przez czas pojedynczego taktu, czyli podzielona przez częstotliwość taktowania, daje oszacowanie czasu rzeczywistego, czasu zegara zewnętrznego w stosunku do systemu komputerowego (tzw. *wall-clock time*). Dla taktów referencyjnych należy ich liczbę (oznaczaną np. jako l_N) podzielić przez częstotliwość nominalną (ν_N), a dla taktów rzeczywistych ich liczbę (l_R) przez częstotliwość rzeczywistą (ν_R), w obu przypadkach obliczony czas zewnętrzny jest taki sam:

$$\frac{l\_N}{\nu\_N} = \frac{l\_R}{\nu\_R}$$

W efekcie można obliczyć uśrednioną częstotliwość rzeczywistą pracy rdzenia jako iloczyn częstotliwości nominalnej i stosunku liczby taktów rzeczywistych do liczby taktów referencyjnych podczas wykonania fragmentu kodu:

$$\nu\_R = \nu\_N \cdot \frac{l\_R}{l\_N}$$

3.9.3 Prosty przykład wykorzystania liczników sprzętowych

Pierwszy prezentowany przykład wykorzystania liczników sprzętowych dotyczy przewidywania rozgałęzień. Badany fragment kodu ma postać:

```
for(i=0;i<1000000;i++)
  x = drand48();
  if(x<0.5) {
    // proste operacje arytmetyczne
  }
}
```

W każdej z miliona wykonywanych pętli znajdują się dwa skoki warunkowe:

- jeden, prosty do przewidzenia, skok na początek pętli związany z warunkiem dotyczącym zmiennej `i` (tylko raz nie wykonany, pozostałe razy wykonany)

- drugi, związany z wartością zmiennej `x`, niemożliwy do przewidzenia (funkcja `drand48` zwraca losową wartość z przedziału $(0, 1)$)

W celu sprawdzenia działania układu przewidywania rozgałęzień, kod, po kompilacji do pliku wykonywalnego *a.out*[6], uruchamiany jest w ramach narzędzia *perf*:

```
$ perf stat a.out
```

Jedną ze zwracanych przez `perf stat` wartości jest wartość licznika związanego ze zdarzeniem niepoprawnego przewidywania rozgałęzień (*branch misses*), która w przypadku badanego kodu osiąga wartość zbliżoną do przewidywanej liczby 500000, połowy wszystkich iteracji:

```
Performance counter stats for 'a.out':
...
        368727897     cycles
...
        500846        branch-misses
...
        0,092846243   seconds time elapsed
```

Zwraca uwagę fakt, że praktycznie wszystkie skoki związane z realizacją pętli są przewidziane prawidłowo (z 2000000 skoków warunkowych w programie, ok. 1500000 jest przewidzianych prawidłowo, z tego ok. 1000000 związanych z realizacją pętli i ok. 500000 z realizacją instrukcji warunkowej *if*). Oznacza to, że układ przewidywania skoków jest w stanie niezależnie uwzględniać co najmniej dwa miejsca wystąpienia skoków w kodzie asemblera.

3.10 Testowanie opóźnienia i przepustowości przetwarzania rozkazów

Testem mierzącym opóźnienie wykonania wybranych rozkazów może być wykonanie przez procesor sekwencji takich rozkazów (także np. w pętli), w przypadku kiedy każdy następny rozkaz jako daną wejściową wykorzystuje daną wyjściową rozkazu poprzedzającego (jest to relatywnie łatwe do uzyskania dla operacji arytmetycznych). W przypadku tak jawnie realizowanej zależności danych, a także w ogólnym przypadku, gdy procesor nie może korzystać ze współbieżności przetwarzania, opóźnienie staje się czynnikiem decydującym o wydajności (w praktyce, dla rozbudowanych programów, sytuacja taka zdarza się rzadko – kompilatory optymalizujące starają się usunąć zależności, a same procesory korzystają np. z przekazywania argumentów pomiędzy rozkazami bezpośrednio w ramach potoków wykonania (*operand forwarding*) oraz innych technik optymalizacji wydajności omawianych w dalszej części rozdziału).

Do badania opóźnienia przetwarzania rozkazów wykorzystany może zostać fragment kodu, zawierający operacje zmiennoprzecinkowe dodawania i mnożenia podwójnej precyzji:

```
for(i=0;i<1000000;i++)
  a = 1.000001*a+0.000001;
}
```

Kompilacja z włączoną opcją optymalizacji `-O3` (szerzej o optymalizujących kompilatorach w p. 5) daje w wyniku następujący kod asemblera (dla kompilatora *gcc*):

[6]Przy kompilacji należy zwrócić uwagę czy kompilator nie zastosował optymalizacji zmieniających charakter wykonywanego kodu, np. dokonując rozwinięcia pętli, zamiany skoku na operacje arytmetyczne lub tp. (patrz p. 5)

```
.L3:
mulsd %xmm0, %xmm1
subl $1, %eax
addsd %xmm2, %xmm1
jne .L3
```

Kompilator (podobne warianty uzyskuje się dla obu stosowanych w pracy kompilatorów *gcc* i *icc*) do wykonywania operacji arytmetycznych zastosował rozkazy wektorowe `mulsd` i `addsd`, operujące na 128-bitowych rejestrach `xmm`. Czas wykonania pętli jest zdeterminowany wyłącznie przez czas realizacji operacji zmiennoprzecinkowych, pozostałe operacje (skok i działanie na indeksie pętli, będącym zmienna całkowitą) wykonywane są na odrębnych potokach przetwarzania, niejako "w tle" operacji zmiennoprzecinkowych (w efekcie optymalizacja kompilatora polegająca na modyfikacji sposobu sprawdzania warunku końca pętli w stosunku do kodu źródłowego, okazuje się być bez znaczenia dla wydajności).

Pomiar czasu wykonania kodu na komputerze wyposażonym w omawiany wcześniej procesor Intel Core i7-4790 (z rdzeniami o architekturze Haswell i nominalną częstotliwością pracy 3.60 GHz) daje wydajność ok. 1 Gflop/s (10^9 operacji zmiennoprzecinkowych na sekundę). Jak widać jest to wydajność kilkadziesiąt razy mniejsza od maksymalnej nominalnej wydajności pojedynczego rdzenia (obliczonej w p. 3.6 jako 57.6 Gflop/s).

Program uruchomiony jest w postaci jednowątkowego procesu na pojedynczym rdzeniu, co pozwala na osiągnięcie zwiększonej częstotliwości pracy rdzenia, ok. 4 GHz. Wyniki wskazują na wydajność średnią CPI=4 (4 takty na pojedynczą operację – bardziej szczegółowe badania wskazują na opóźnienie 3 takty dla `addsd` i 5 taktów dla `mulsd`). Co powoduje tak niską wydajność przetwarzania?

Kod napisany jest w taki sposób (co dobrze widać analizując postać asemblera), że w każdej operacji zmiennoprzecinkowej wykorzystywany jest ten sam rejestr zawierający modyfikowaną wartość zmiennej *a* (w konkretnym przypadku użytych kompilatorów jest to rejestr `xmm1`). Oznacza to, że kolejny rozkaz zmiennoprzecinkowy w kodzie nie może zostać wykonany, dopóki nie zostanie zakończony rozkaz poprzedzający. Procesor/rdzeń nie jest w stanie w pełni ujawnić swoich możliwości przetwarzania związanych z technikami ukrywania opóźnienia (*latency hiding*): przetwarzaniem potokowym i wieloma potokami przetwarzania rozkazów.

W celu wykrycia maksymalnej wydajności (przepustowości) przetwarzania pojedynczego rdzenia, testowy kod jest modyfikowany:

```
for(i=0;i<1000000;i++)
  a = 1.000001*a+0.000001;
  b = 1.000001*b+0.000001;
  c = 1.000001*c+0.000001;
  // itd. dla dalszych zmiennych
}
```

Dodawanie kolejnych zmiennych w kodzie pozwala na wykorzystanie większej liczby potoków oraz sprawniejsze przetwarzanie potokowe, kiedy każdy potok może współbieżnie przetwarzać nie powiązane ze sobą rozkazy dotyczące różnych zmiennych. Zgodnie z prawem Little'a, można dokonać obliczenia ile zmiennych w pętli potrzeba, aby uzyskać maksymalną wydajność, a więc pełne wykorzystanie potoków przetwarzania. Jeśli opóźnienie przetwarzania pojedynczego rozkazu przez pojedynczy potok wynosi 4 takty, a rdzeń posiada dwa potoki przetwarzania rozkazów zmiennoprzecinkowych, to wymagana liczba zmiennych, czyli niezależnych rozkazów w każdej iteracji pętli, wynosi $2 \cdot 4 = 8$.

Eksperymentalne pomiary czasu wykonania wskazują, że dodanie każdej zmiennej zwiększa wydajność przetwarzania o ok. 1 Gflop/s, osiągając wydajność ok. 7.2 Gflop/s dla 8 zmiennych. Uwzględniając częstotliwość pracy rdzenia daje to przyrost ok. 0.25 IPC (jako odwrotność CPI=4) dla każdej nowej

zmiennej i w ostateczności, dla 8 zmiennych, prawie dwa rozkazy kończone w każdym takcie. Dodanie jeszcze dwóch dodatkowych zmiennych pozwala w pełni nasycić potoki, prowadząc do wartości IPC równej 1,99 i wydajności 7,97 Gflop/s. Można uznać to za osiągniętą w praktyce z dużą dokładnością teoretyczną maksymalną wydajność przetwarzania rozkazów zmiennoprzecinkowych dodawania i mnożenia przez dwa potoki rdzenia.

Wydajność taka wciąż jest daleka od maksymalnej wydajności rdzenia. Jedną z wad dotychczasowego rozwiązania jest użycie odrębnych rozkazów dodawania i mnożenia, podczas gdy rdzeń potrafi wykonać połączone dodawanie i mnożenie w pojedynczym rozkazie FMA (*fused multiply-add*). Zamiana odrębnych rozkazów na pojedynczy połączony rozkaz następuje po przekazaniu do kompilatora opcji jawnie wskazującej na typ architektury rdzenia, w tym wypadku opcji: `-march=core-avx2`.

Po zastosowaniu opcji, wydajność dla pojedynczej zmiennej w pętli wynosi ok. 1,6 Gflop/s, co przy częstotliwości pracy rdzenia 4 GHz, odpowiada opóźnieniu IPC=5 taktów na pojedynczy rozkaz (w efekcie IPC=1/CPI=0,2). Dodawanie kolejnych zmiennych powoduje wzrost parametru IPC o ok. 0,2, aż do osiągnięcia wartości zbliżonej do 2 dla 10 zmiennych. Wydajność w tym ostatnim przypadku można uznać za miarę przepustowości potoków przetwarzania dla skalarnej operacji FMA. Wynosi ona w praktyce ok. 15.75 Gflop/s, co odbiega tylko o mniej niż 2% od teoretycznej przepustowości 16 Gflop/s (2 rozkazy FMA, czyli 4 operacje zmiennoprzecinkowe, kończone w każdym takcie).

Kolejnym brakiem powstałego kodu jest działanie skalarne. Wprawdzie w asemblerze pojawiają się rejestry i instrukcje wektorowe, ale w każdym rejestrze 128-bitowym znajduje się tylko jedna liczba podwójnej precyzji i operacja na takim rejestrze jest efektywnie tylko jedną operacją zmiennoprzecinkową. Uzyskane dotychczas parametry charakteryzują więc możliwości skalarnego przetwarzania operacji zmiennoprzecinkowych przez rdzeń.

W celu umożliwienia kompilatorowi pełnego wykorzystania możliwości przetwarzania wektorowego konieczne jest dokonanie dalszych modyfikacji kodu. Zamiast pojedynczych zmiennych wykorzystane zostają małe tablice o rozmiarze 16:

```
for(i=0;i<1000000;i++)
  for(k=0; k<16; k++){
    a_tab[k] = 1.000001*a_tab[k]+0.000001;
    b_tab[k] = 1.000001*b_tab[k]+0.000001;
    c_tab[k] = 1.000001*c_tab[k]+0.000001;
    // itd. dla dalszych zmiennych
  }
}
```

Już użycie pojedynczej tablicy zwiększa wydajność przetwarzania do ok. 25 Gflop/s. Wynika to z pełnego wykorzystania czterech 256-bitowych rejestrów AVX do przechowywania całej tablicy, co pozwala współbieżnie wykonywać cztery operacje FMA na rejestrach. Dla rdzeni Haswell opóźnienie rozkazów wektorowych FMA, podobnie jak rozkazów skalarnych, wynosi 5 taktów, czyli IPC jest równe 0.2. Dla czterech współbieżnie wykonywanych rozkazów IPC wynosi 0.8, co oznacza kończenie 3.2 skalarnych operacji FMA na pojedynczy takt. Użycie dwóch tablic zwiększa wydajność do ok. 51 Gflop/s (IPC=1,6, przy częstotliwości pracy rdzenia ok. 4 GHz), a najwyższą praktycznie uzyskiwaną wydajność, 63 Gflop/s, zapewnia wykorzystanie 3 tablic (dokładnie jak w przedstawionym kodzie). Wydajność taka odpowiada ok. 16 skalarnym operacjom arytmetycznym w pojedynczym takcie (co jest równoważne IPC=2 dla wektorowych rozkazów FMA w pojedynczym rdzeniu) – zgodnie z przewidywaniami teoretycznymi.

Uzyskana wcześniej nominalna maksymalna wydajność pojedynczego rdzenia analizowanego mikroprocesora, obliczana na podstawie danych producenta, wynosi 57.6 Gflop/s (3.6 GHz x 16 operacji zmiennoprzecinkowych w pojedynczym takcie, jako efekt wykorzystania 2 potoków 256-bitowych

rozkazów wektorowych FMA). Zwiększenie do wartości uzyskanej eksperymentalnie (dla trzech tablic i przetwarzania wektorowego) związane jest z podwyższeniem częstotliwości przetwarzania do ok. 4 GHz, możliwym w przypadku użycia wyłącznie jednego rdzenia.

Uruchomienie powyższego kodu w postaci wielowątkowej (np. z wykorzystaniem biblioteki wątków POSIX, *pthreads*) pozwala osiągnąć na 4 rdzeniach wydajność ok. 240 Gflop/s (przy częstotliwości pracy ok. 3,8 GHz).

W obu przypadkach, jedno i wielowątkowym, wydajność przetwarzania uzyskana eksperymentalnie jest zbliżona (z dokładnością do ok. 2%) do teoretycznej maksymalnej wydajności 16 operacji zmiennoprzecinkowych kończonych w pojedynczym takcie przez pojedynczy rdzeń mikroprocesora.

Warto jeszcze zwrócić uwagę na płynący z powyższego przykładu wniosek o relatywnym znaczeniu współczynnika CPI, dotyczący nie tylko różnicy między rozkazami skalarnymi i wektorowymi. Obserwacja, że kończenie w każdym takcie jednej operacji skalarnej (CPI=1) oznacza kilkukrotnie niższą wydajność niż kończenie jednej pełnowartościowej operacji wektorowej (CPI także równe jeden), jest wnioskiem słusznym i oczywistym. Mniej oczywisty jest fakt, że kompilator może użyć operacji (potoków) przetwarzania wektorowego w przypadku nie w pełni wykorzystanych rejestrów wektorowych. Wtedy wartość współczynnika CPI (nawet uwzględniając, że dotyczy rozkazów wektorowych) ponownie nie odpowiada rzeczywistej wydajności programu – istotnym staje się obsadzenie rejestrów użytecznymi danymi programu i efektywna liczba operacji kodu źródłowego realizowanych w pojedynczym rozkazie wektorowym.

3.10.1 ”Ciśnienie na rejestry” i rozdzielanie pętli

Ciekawym faktem związanym z przedstawionym w poprzednim punkcie kodem jest gwałtowne zmniejszenie wydajności przetwarzania w przypadku użycia czterech tablic.

```
for(i=0;i<1000000;i++)
  for(k=0; k<16; k++){
    a_tab[k] = 1.000001*a_tab[k]+0.000001;
    b_tab[k] = 1.000001*b_tab[k]+0.000001;
    c_tab[k] = 1.000001*c_tab[k]+0.000001;
    d_tab[k] = 1.000001*d_tab[k]+0.000001;
  }
}
```

Kod asemblera dla trzech tablic wygląda następująco (tym razem dla kompilatora *icc*):

```
..B1.6:
        vfmadd213pd %ymm0, %ymm13, %ymm12
        incl      %eax
        vfmadd213pd %ymm0, %ymm13, %ymm11
        vfmadd213pd %ymm0, %ymm13, %ymm10
        vfmadd213pd %ymm0, %ymm13, %ymm9
        vfmadd213pd %ymm0, %ymm13, %ymm8
        vfmadd213pd %ymm0, %ymm13, %ymm7
        vfmadd213pd %ymm0, %ymm13, %ymm6
        vfmadd213pd %ymm0, %ymm13, %ymm5
        vfmadd213pd %ymm0, %ymm13, %ymm4
        vfmadd213pd %ymm0, %ymm13, %ymm3
        vfmadd213pd %ymm0, %ymm13, %ymm2
```

```
        vfmadd213pd %ymm0, %ymm13, %ymm1
        cmpl        $10000000, %eax
        jb          ..B1.6
```

Podczas gdy dla czterech tablic otrzymuje się:

```
..B1.7:
        vmovupd   64(%rsp,%rdx,8), %ymm2
        vmovupd   192(%rsp,%rdx,8), %ymm3
        vmovupd   320(%rsp,%rdx,8), %ymm4
        vmovupd   448(%rsp,%rdx,8), %ymm5
        vfmadd213pd %ymm0, %ymm1, %ymm2
        vfmadd213pd %ymm0, %ymm1, %ymm3
        vfmadd213pd %ymm0, %ymm1, %ymm4
        vfmadd213pd %ymm0, %ymm1, %ymm5
        vmovupd   %ymm2, 64(%rsp,%rdx,8)
        vmovupd   %ymm3, 192(%rsp,%rdx,8)
        vmovupd   %ymm4, 320(%rsp,%rdx,8)
        vmovupd   %ymm5, 448(%rsp,%rdx,8)
        addq      $4, %rdx
        cmpq      $16, %rdx
        jb        ..B1.7
```

W drugim przypadku, kompilatorowi brakuje rejestrów, żeby w nich umieścić wszystkie zmienne występujące w pojedynczej iteracji, i w efekcie, zamiast efektywnego przetwarzania z wykorzystaniem wyłącznie rejestrów, procesor realizuje w każdej iteracji 8 dostępów do pamięci (za pomocą wektorowych wariantów rozkazu *mov*), co nawet w przypadku wyłącznego korzystania z pamięci podręcznej najbliższej potokom przetwarzania, znacznie spowalnia wykonanie programu.

Powyższe negatywne zjawisko, użycia wewnątrz pętli zbyt wielu zmiennych, uniemożliwiające efektywne korzystanie z rejestrów, zwane jest ”ciśnieniem na rejestry” (*register pressure*). Unikanie ”ciśnienia na rejestry” jest jednym z wskazań przy tworzeniu wysoko wydajnego kodu.

Dla konkretnego rozważanego przykładu, techniką optymalizacji pozwalającą na likwidację ”ciśnienia na rejestry” i osiągnięcie wyższej wydajności jest rozdzielenie pętli (*loop fission*). Przeprowadzenie rozdzielenia pętli z indeksem i na dwie odrębne pętle:

```
for(i=0;i<1000000;i++){
  for(k=0; k<16; k++){
    a_tab[k] = 1.000001*a_tab[k]+0.000001;
    b_tab[k] = 1.000001*b_tab[k]+0.000001;
  }
}
for(i=0;i<1000000;i++){
  for(k=0; k<16; k++){
    c_tab[k] = 1.000001*c_tab[k]+0.000001;
    d_tab[k] = 1.000001*d_tab[k]+0.000001;
  }
}
```

nie powoduje zmiany wyniku (rozdzielane obliczenia są od siebie całkowicie niezależne), a pozwala na podniesienie wydajności z ok. 28 Gflop/s (kod z czterema tablicami w każdej iteracji wewnętrznej pętli)

do ok. 50 Gflop/s (kod z dwiema tablicami w każdej wewnętrznej iteracji). W przypadku sześciu tablic, *loop fission* pozwala na zwiększenie wydajności z ok. 30 Gflop/s (wszystkie sześć tablic w jednej pętli z indeksem *k*) do ok. 63 Gflop/s po rozdzieleniu na dwie podwójne pętle. Oznacza to ponad dwukrotne skrócenie czasu wykonania całości obliczeń.

3.10.2 "Rozciąganie" tablic w celu optymalizacji przetwarzania wektorowego

Wyobraźmy sobie sytuację, kiedy z wymagań aplikacji wynika, że w każdej z tablic `a_tab`, `b_tab` i `c_tab` mamy do przechowania 15 egzemplarzy danych, na których mamy wykonać operacje, takie jak w dotychczasowym kodzie. Naturalne zaprojektowanie kodu z rozmiarem tablic 15 i taką samą liczbą operacji w pętli wewnętrznej, powoduje zmniejszenie wydajności przetwarzania w wersji wielowątkowej do ok. 75 Gflop/s. Analiza kodu asemblera wyprodukowanego przez kompilator (*icc*) pokazuje, że zastosował on, podobnie jak w przypadku ciśnienia na rejestry, dostępy do pamięci, zamiast czystego przetwarzania z użyciem rejestrów (dodatkowo zamiast wyłącznie rejestrów 256-bitowych, użył rejestrów 128-bitowych wypełnionych tylko pojedynczymi liczbami podwójnej precyzji).

W celu przywrócenia przetwarzania zbliżonego do optymalnego można w tym momencie zastosować technikę rozciągania (rozpychania) tablic (*array padding*), omówioną w p. 2.1.1. Będzie ona polegać na użyciu tablic większych niż wymaga tego aplikacja, tak aby umożliwić kompilatorowi i sprzętowi działanie prowadzące do wyższej wydajności.

Zastosowanie rozciągania tablic jest w tym przypadku niezwykle proste. Należy zaalokować tablice o rozmiarze 16 i zwiększyć liczbę iteracji w pętli wewnętrznej także do 16. Oznacza to wykonywanie operacji także dla 16-go elementu w tablicach, który dobrze jest zainicjować (np. wartością zero), tak aby uniknąć ewentualnych (choć mało prawdopodobnych) problemów przetwarzania wartości nietypowych (np. NaN).

Po kompilacji i uruchomieniu okazuje się, że czas wykonania programu uległ znacznemu skróceniu, mimo że w całym programie wykonywana jest większa liczba operacji niż przed optymalizacją.

W konsekwencji konieczne jest zmodyfikowanie sposobu liczenia wydajności. Mimo wykonywania 16 iteracji w pętli wewnętrznej, efektywna praca na potrzeby aplikacji dotyczy tylko 15 iteracji i 15 elementów każdej z tablic. Co oznacza, że praca sprzętu z wydajnością ok. 240 Gflop/s (uzyskana w układzie optymalnym), na potrzeby aplikacji daje tylko ok. 227 Gflop/s (czyli ok. 240*15/16). Ta ostatnia wartość jest przyjmowana jako efektywna wydajność, zgodnie z konwencją przyjętą w książce.

3.10.3 Zestawienie wyników charakteryzujących wydajność potoków zmiennoprzecinkowych pojedynczego rdzenia mikroprocesora Intel Core i7-4790 o architekturze Haswell, pracującego z częstotliwością 4 GHz

Tabela 3.1 przedstawia zastawienie szeregu wyników wydajnościowych uzyskanych w opisywanych w niniejszym rozdziale testach przetwarzania potokowego rozkazów zmiennoprzecinkowych. Kolejne wiersze tabeli można interpretować jako sprawdzanie w praktyce teoretycznych możliwości potoków, dla zmiennych podwójnej precyzji (64-bitowych). Kolumny tabeli pokazują uzyskaną wydajność w Gflop/s, uśrednioną wartość parametru IPC dla użytych rozkazów oraz wydajność w liczbie zmiennoprzecinkowych operacji arytmetycznych na pojedynczy takt pracy procesora [flop/cycle]. Obliczenia wartości z dwóch ostatnich kolumn oparte są o pomiar częstotliwości pracy rdzenia w trakcie obliczeń, który każdorazowo dawał w wyniku 4 GHz, z dokładnością wyższą od 1%.

Pierwsza grupa wierszy w tabeli dotyczy przetwarzania skalarnego (rozkazy dotyczą rejestrów skalarnych lub rejestrów wektorowych, w których wykorzystywana jest tylko jedna pozycja obsadzona przez zmienną 64-bitową). Pierwszy wiersz, odpowiadający pomiarowi opóźnienia wykonywania rozkazów, dotyczy sytuacji wykonywania rozkazów przez pojedynczy potok, kiedy na skutek zależności w kodzie,

Opis przypadku	Wydajność [Gflop/s]	Uśrednione IPC	Wydajność [flop/cycle]
przetwarzanie skalarne 1-zmienna (bez FMA)	0,99	0,25	0,25
przetwarzanie skalarne 2-zmienne (bez FMA)	1,98	0,50	0,50
przetwarzanie skalarne 8-zmiennych (bez FMA)	7,21	1,80	1,80
przetwarzanie skalarne 10-zmiennych (bez FMA)	7,97	1,99	1,99
przetwarzanie skalarne 16-zmiennych (bez FMA)	6,33	1,58	1,58
przetwarzanie skalarne 1-zmienna (FMA)	1,59	0,2	0,4
przetwarzanie skalarne 8-zmiennych (FMA)	12,56	1,57	3,14
przetwarzanie skalarne 10-zmiennych (FMA)	15,86	1,98	3,96
przetwarzanie skalarne 16-zmiennych (FMA)	7,50	0,93	1,87
przetwarzanie wektorowe 1-tablica 4-elementowa (FMA)	6,34	0,2	1,58
przetwarzanie wektorowe 1-tablica 8-elementowa (FMA)	12,70	0,4	3,17
przetwarzanie wektorowe 1-tablica 40-elementowa (FMA)	62,52	1,96	15,64
przetwarzanie wektorowe 1-tablica 48-elementowa (FMA)	62,97	1,97	15,75
przetwarzanie wektorowe 1-tablica 16-elementowa (FMA)	25,39	0,79	6,35
przetwarzanie wektorowe 2-tablice 16-elementowe (FMA)	50,71	1,58	12,68
przetwarzanie wektorowe 3-tablice 16-elementowe (FMA)	63,35	1,98	15,84
przetwarzanie wektorowe 4-tablice 16-elementowe (FMA)	28,23	0,88	7,06

Tablica 3.1: Wyniki wydajnościowe przetwarzania potokowego dla pojedynczego rdzenia mikroprocesora Intel Core i7-4790

w którym jest tylko jedna zmienna, nie występuje przetwarzanie współbieżne, rozkazy wykonywane są sekwencyjnie, jeden po drugim. Dodatkowo w tej grupie, brak użycia odpowiednich opcji kompilatora powoduje, że nie są stosowane rozkazy FMA.

Kolejne wiersze prezentują wyniki dla przypadków, gdy na skutek zwiększania liczby zmiennych w kodzie, a więc zwiększania liczby niezależnych instrukcji w pojedynczej iteracji pętli kodu testującego, rośnie stopień współbieżności działania potoków. Stopień ten zbliża się do teoretycznego maksimum, kiedy w każdym takcie kończone są dwa rozkazy zmiennoprzecinkowe, przez dwa potoki rdzenia, dla 8 zmiennych podwójnej precyzji. Wynika to bezpośrednio z opóźnienia przetwarzania rozkazów arytmetycznych mnożenia i dodawania, wynoszącego średnio 4 takty (dokładnie 3 takty dla `addsd` i 5 taktów dla `mulsd`). Dalsze dodawanie zmiennych może prowadzić do niewielkiego wzrostu wydajności i zbliżania się do teoretycznego maksimum. W pewnym momencie należy spodziewać spadków wydajności spowodowanych zjawiskiem ciśnienia na rejestry (tak jest w umieszczonym w tabeli przypadku 16 zmiennych).

Druga grupa wierszy tabeli 3.1 odpowiada sytuacji analogicznej jak w przypadku pierwszej grupy, z jedyna różnica polegająca na zastosowaniu opcji kompilacji powodujących użycie rozkazów FMA (*fused multiply-add*) w kodzie binarnym. Wyniki pokazują jak rośnie wydajność, od minimalnej dla sytuacji testowania opóźnienia przetwarzania (1 zmienna, brak współbieżności), do maksymalnej w przypadku testowania przepustowości potoków (10 zmiennych, pełna współbieżność pracy dla każdego z dwóch potoków rdzenia). Wyniki pokazują, jak użycie rozkazów FMA, przy wzroście parametru IPC (od 0,2 do 2 rozkazów na takt) prowadzi do wyższej wydajności w Glop/s i [flop/cycle], w porównaniu dla przypadku bez użycia rozkazu FMA. W przypadku 1 zmiennej przyrost ten jest mniejszy niż oczekiwany dwukrotny, ze względu na większe opóźnienie w stosunku do uśrednionego opóźnienia rozkazów dodawania i mnożenia. Jednak w momencie, kiedy zgodnie z prawem Little'a udaje się w pełni nasycić

potoki przetwarzania, wydajność FMA staje sie dwukrotnie wyższa od wydajności mnożenia i dodawania, dzięki tej samem przepustowości 2 rozkazów na takt. Ponownie, ostatni wynik w tej grupie wierszy, dla 16 zmiennych podwójnej precyzji, wskazuje na spadek wydajności, na skutek zjawiska ciśnienia na rejestry.

Ostatnia, trzecia grupa wierszy odnosi się do przypadku użycia opcji kompilacji prowadzących do pełnej wektoryzacji kodu, łącznie z zastosowaniem rozkazów FMA. Kolejne wiersze odpowiadają coraz większej liczbie zmiennych przetwarzanych wektorowo w pojedynczej iteracji pętli testującej (najpierw poprzez użycie coraz dłuższej tablicy, a potem przez użycie wielu tablic), co prowadzi do rosnącego stopnia współbieżności przetwarzania. Dla liczby iteracji w pętli od 4 do 40, parametr IPC zwiększa się w kolejnych wierszach od wartości 0,2 do 2, co jest bezpośrednią konsekwencją faktu, że wektorowe rozkazy FMA dla rejestrów 256 bitowych, podobnie jak rozkazy skalarne, mają opóźnienie 5 taktów. Dzięki spakowaniu 4 zmiennych do jednego rejestru 256-bitowego wydajność mierzona w flop/cycle rośnie od ok. 1,6 do ok. 16, a wyrażana w Gflop/s od ok. 6,3 do ok. 63 (dzieki uzyskanej częstotliwości pracy 4 GHz).

Ostatni wiersz tabeli pokazuje nagły spadek wydajności dla sytuacji użycia odniesień do pamięci w kodzie binarnym pojedynczej iteracji pętli testującej, w efekcie zjawiska ciśnienia na rejestry. W tym przypadku spadek jest znacznie bardziej znaczący, niż dla przetwarzania skalarnego bez rozkazów FMA.

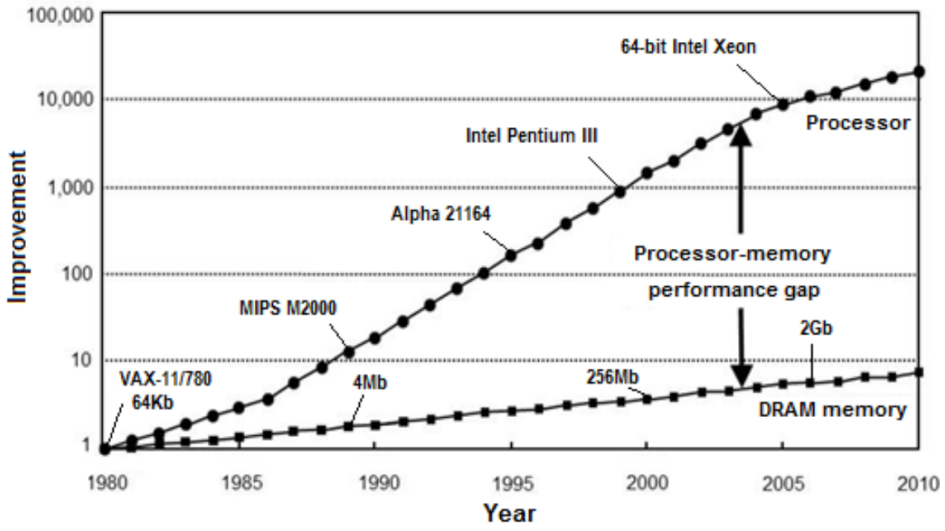
Rozdział 4

Model i wydajność dostępów do pamięci dla obliczeń jednowątkowych

Pojedynczy wątek pracujący na pojedynczym rdzeniu współczesnego mikroprocesora realizując dostępy do zmiennych korzysta z hierarchii poziomów pamięci. Zmiennej w kodzie źródłowym odpowiada nazwany obszar pamięci operacyjnej (pamięci głównej – *primary storage, main memory*), najczęściej fizycznie realizowanej w technologii DRAM (*dynamic random access memory*), stąd dalej często używane będzie określenie pamięć DRAM na oznaczenie pamięci głównej. W rozkazach asemblera jako argumenty, obok zawartości rejestrów i stałych będących argumentami bezpośrednimi, używane są adresy zmiennych w pamięci operacyjnej (zapisane w rejestrach lub obliczone na podstawie zawartości rejestrów, w złożonych trybach adresowania, omawianych w p. 3.2).

Od wielu lat obserwowane jest zjawisko znacznie wolniejszego przyrostu wydajności modułów pamięci DRAM w stosunku do wydajności procesorów (rdzeni), tzw. *memory wall*. Szczególnie jest to widoczne w odniesieniu do opóźnienia jako miary wydajności, bez uwzględnienia szeregu technik ukrywania opóźnienia przy dostępie do pamięci (*memory latency hiding*), o których będzie mowa w dalszej części książki. Zjawisko *memory wall* ilustruje rys. 4.1, pokazujacy jak od początku lat 80-tych XX wieku, przyjętych jako punkt odniesienia, różnica w wydajności między procesorami CPU a modułami DRAM powiększyła się ponad tysiąc razy.

Zjawisko *memory wall* stało się motywacją do zastosowania pośrednich poziomów pamięci, z wykorzystaniem szybszej technologii SRAM (*static random access memory*). Powstała w ten sposób hierarchia pamięci, począwszy od rejestrów, poprzez pośrednie poziomy pamięci podręcznej (*cache memory*), najczęściej realizowane właśnie w technologii SRAM, aż do pamięci głównej DRAM, a nawet, poprzez mechanizm pamięci wirtualnej, do pamięci zewnętrznej (*external memory*). Każdy kolejny poziom w tej hierarchii charakteryzuje się niższą wydajnością, ale w zamian za to także mniejszym kosztem jednostkowym (na pojedynczy przechowywany bajt) i w związku z tym większą pojemnością efektywnie umieszczaną w układach procesora i całego komputera. Ilustruje to rys. 4.2, na którym poza poziomami pamięci wykorzystywanymi do przechowywania danych w trakcie wykonywania programu znajdują się także pamięci stosowane wyłącznie do archiwizacji danych i nowy proponowany poziom, oznaczany jako *Persistent memory*, realizowany np. w jednym z rozwijanych obecnie wariantów technologii NVRAM (NRAM, MRAM itp.). Ilustracja, poza zaznaczeniem pojemności (*capacity*), kosztu (*cost*) i czasu dostępu do danych (*latency*) dla każdego poziomu pamięci, wskazuje na różnice w trwałości zapisanych danych (pamięci ulotne, *volatile*, i nieulotne, *non-volatile*), w sposobie dostępu (za pomocą rozkazów procesora, *load/store instructions* i poleceń systemu operacyjnego, *I/O commands*), a także w jednostkowym rozmiarze danych (*granularity*) przy praktycznej realizacji dostępu (dostęp do linii pamięci podręcznej, *cache line* i do bloku pamięci zewnętrznej). Największe pojemności oferują: pamięć



Rysunek 4.1: Przyrost wydajności procesorów (rdzeni) i modułów pamięci DRAM w latach 1980-2010 [źródło: Wikipedia].

zewnętrzna (drugiego rzędu, *secondary storage*), najczęściej występująca jako twarde dyski HDD lub układy SSD[1], gdzie pojemności sięgają terabajtów pamięci, oraz taśmy magnetyczne, o pojemnościach rzędu petabajtów.

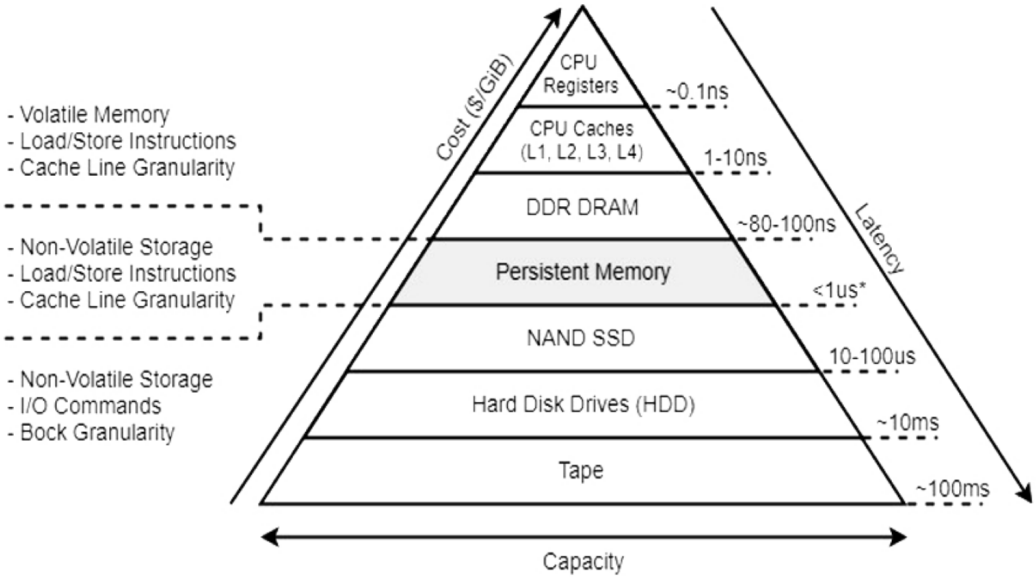
We współczesnych systemach komputerowych stosuje się kilka (najczęściej 2 lub 3, rzadziej 4) poziomy pamięci podręcznej (poziom czwarty bywa wykonany w technologi eDRAM). W pamięciach podręcznych, do których nie ma bezpośredniego dostępu z poziomu kodu źródłowego w standardowych językach programowania, przechowuje się kopie wartości zmiennych z pamięci głównej. Zasadą organizacji pamięci podręcznej jest umieszczanie, w miarę oddalania się od potoków przetwarzania, kolejnych poziomów pamięci (L1, L2, L3 - *level 1, level 2, level 3*), z których każdy następny charakteryzuje się mniejszą szybkością działania[2].

W ramach omawiania hierarchii pamięci w niniejszej książce, uwzględnione będą tylko pamięć główna i pamięć podręczna. Pamięć zewnętrzna jest zazwyczaj o co najmniej jeden rząd wielkości wolniejsza od pamięci DRAM, stąd przy optymalizacji wydajności należy unikać konieczności jej użycia. Istnieje szereg aplikacji, w których korzystanie z pamięci zewnętrznej jest jednak konieczne, np. ze względu na rozmiar używanych struktur danych. W takich wypadkach, przy optymalizacji wydajności przydatne mogą być niektóre z technik opisywanych w dalszych punktach dla pamięci DRAM i pamięci podręcznych (np. zwiększanie lokalności odniesień w programie), często też optymalizacja będzie istotnie zależna od specyfiki aplikacji (np. od rodzaju konkretnych struktur danych użytych do przechowywania zmiennych aplikacji)[3].

[1] W dalszej części ze względu na popularność twardych dysków, określenie to będzie czasami zamiennie używane z określeniem pamięć zewnętrzna.

[2] Mimo oznaczania kolejnych poziomów pamięci podręcznej symbolami L1, L2, L3, co można przyjąć za hierarchię pamięci od najniższego poziomu do najwyższego, funkcjonuje także konwencja uznawania pamięci L1 za poziom najwyższy, a kolejne za coraz niższe. W niniejszej książce przyjęta jest konwencja określania poziomów jako znajdujących się bliżej lub dalej od potoków przetwarzania (z L1 jako poziomem najbliższym). Ostatni poziom przed pamięcią DRAM (zazwyczaj L3, ale bywa także L2 lub L4), oznaczany często jako LLC (*last level cache)*, jest w takim ujęciu poziomem najdalszym (nazywany będzie dalej także poziomem ostatnim).

[3] Specjalne algorytmy, nieopisywane w niniejszej pracy, korzystające z pamięci zewnętrznej (jawnie, bez pośrednictwa mechanizmu pamięci wirtualnej, i z założenia w sposób dążący do optymalnego) nazywane są algorytmami *out-of-core* (*out-of-core algorithms*).



Rysunek 4.2: Hierarchia pamięci współczesnych systemów komputerowych [źródło: Wikipedia].

Jednym z aspektów wpływających na wydajność dostępów do pamięci jest charakter dostępu, czy jest to odczyt, zapis czy modyfikacja wartości zmiennych, oznaczająca pobranie i zapis. W tym i następnym rozdziale badania prowadzone są głównie w kontekście odczytów z pamięci, choć w większości dotyczą także innych rodzajów dostępu. Zagadnienia specyficzne dla modyfikacji i zapisów danych analizowane są bardziej szczegółowo w rozdziałach omawiających funkcjonowanie pamięci w programach wielowątkowych.

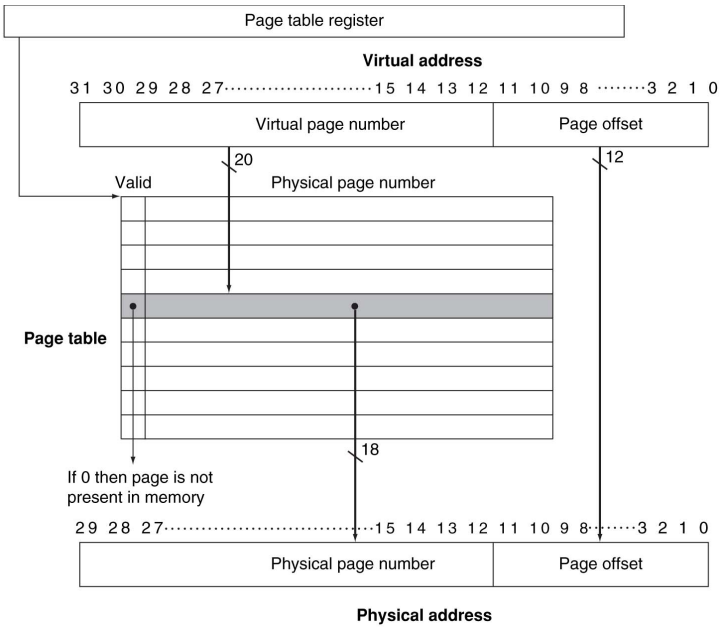
4.1 Pamięć wirtualna

Pierwszym z mechanizmów, który zostanie omówiony w kontekście projektowania i optymalizacji korzystania z danych w pamięci operacyjnej, jest mechanizm pamięci wirtualnej ze stronicowaniem, stosowany przez praktycznie wszystkie współczesne systemy operacyjne ogólnego przeznaczenia.

Przykładowy rozkaz dostępu do pamięci, zapisany w języku asemblera zgodnie z omówionymi wcześniej konwencjami dla procesorów z rodziny $x86$, może wyglądać następująco:

```
mov -4(%ebx,%ecx,2), %eax
```

Oznacza on pobranie wartości zmiennej przechowywanej w pamięci, w komórkach o adresie początkowym obliczonym zgodnie z zasadami złożonych trybów adresowania na podstawie zawartości rejestrów `%ebx` oraz `%ecx` (patrz p. 3.2), i zapisanie jej w rejestrze `%eax` (liczbę pobranych bajtów określa dodatkowa litera dodana na końcu nazwy rozkazu, np. `movl` oznacza pobranie 4 bajtów). Obliczony adres jest adresem wirtualnym, mieszczącym się w zakresie charakteryzującym dany procesor i tryb jego pracy. Zakres ten określa przestrzeń adresową, czyli zbiór adresów dostępnych programowi w trakcie wykonania. W przypadku współczesnych procesorów rodziny $x86$ podstawowy zakres wynosi od 0 do $2^n - 1$, z jednostką adresowania w postaci pojedynczego bajtu i n równym 32 lub 64, w zależności od typu rejestrów używanych do obliczania adresu (w powyższym przykładzie użyto rejestrów 32-bitowych). W praktyce zakres może być modyfikowany, a procesor może udostępniać kilka zakresów, np. poprzez sztuczne ograniczenie przestrzeni adresowej dla procesorów 64-bitowych.



Rysunek 4.3: Wykorzystanie tablicy stron do obsługi pamięci wirtualnej [źródło: Wikipedia].

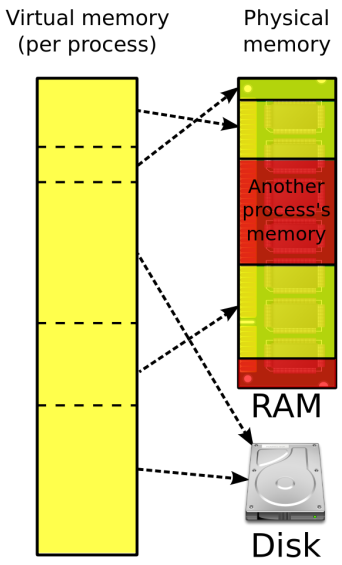
Rozmiar fizycznej pamięci DRAM współczesnych systemów komputerowych jest praktycznie zawsze mniejszy niż dopuszczalne wirtualne maksimum[4]. W mechanizmie pamięci wirtualnej ze stronicowaniem, całość przestrzeni adresowej (pamięci wirtualnej) jest dzielona na strony o określonym rozmiarze (system operacyjny pozwala na wybór rozmiaru strony, z najczęściej stosowaną wartością 4 kB). Pojedynczej stronie (*virtual page*) odpowiada pojedyncza ramka w pamięci DRAM (*physical page*), oznaczająca układ fizycznie przechowujący dane. W pamięci DRAM nie jest przechowywana cała przestrzeń adresowa procesu (wykonywanego programu), ale tylko wybrane jej strony. Jest to naturalną konsekwencją faktu, że przeciętne programy używają dla swojego kodu i danych tylko drobnego ułamka dostępnej przestrzeni adresowej.

Adres wirtualny jest tłumaczony na adres fizyczny związany z konkretną lokalizacją w modułach pamięci DRAM. Obliczenie adresu fizycznego składa się z kilku kroków. Adres wirtualny (*virtual address*) jest podzielony na sekcje o określonej liczbie bitów. W najprostszym przypadku są dwie sekcje: jedna służąca do określenia numeru strony (*virtual page number*) oraz druga dla wewnętrznego adresu w ramach strony (*page offset*). Uzyskany numer strony służy do znalezienia lokalizacji odpowiadającej ramki pamięci DRAM (*physical page number*). W tym celu stosowana jest struktura danych nazywana tablicą stron (*page table*). Informacja przechowywana w tablicy stron wskazuje czy zawartość strony przechowywana jest w ramce pamięci DRAM, i jeśli tak, to zawiera także fizyczny adres (*physical address*) odpowiadającej ramki (rys. 4.3).

W przypadku, gdy program (lub kilka współbieżnie wykonywanych w systemie programów) używa więcej pamięci niż jest dostępne w modułach DRAM, zawartość niektórych stron pamięci wirtualnej jest usuwana z pamięci głównej i zapisywana w pamięci zewnętrznej. Gdy zawartość strony jest przechowywana na twardym dysku, dostęp do żądanego adresu wirtualnego wymaga najpierw pobrania zawartości strony do pamięci DRAM, a dopiero potem możliwy jest dostęp do konkretnej komórki pamięci.

Rysunek 4.4 przedstawia prostą ilustrację mechanizmu działania pamięci wirtualnej z wykorzystaniem pamięci zewnętrznej. Część stron z pamięci wirtualnej wykonywanego programu (procesu) prze-

[4]Historycznie, gdy dominowały procesory 32-bitowe, oznaczało to ok. 4 GB pamięci (co dzisiaj jest coraz rzadziej spotykane), natomiast granica dla procesorów 64-bitowych 16384 PB (petabajtów, 10^{15}B) nie jest dzisiaj praktycznie osiągalna.



Rysunek 4.4: Mechanizm działania pamięci wirtualnej, odwzorowanie sekwencji stron pamięci wirtualnej procesu w zbiór ramek pamięci DRAM, połączone z przechowywaniem zawartości niektórych stron na twardym dysku [źródło: Wikipedia].

chowywana jest w pamięci DRAM, a część na twardym dysku. Rysunek pokazuje także, jak pamięć DRAM wykorzystywana jest jednocześnie przez inny, współbieżny proces – mechanizm pamięci wirtualnej jest wygodnym sposobem separacji obszarów pamięci przyporządkowanych różnym procesom i zapewnienia bezpieczeństwa wykonania wielu współbieżnych procesów w ramach systemów wielozadaniowych.

Zarządzaniem pamięcią wirtualną, w szczególności wykorzystaniem pamięci zewnętrznej, drugiego rzędu, do przechowywania stron pamięci wirtualnej usuniętych z pamięci głównej, zajmuje się system operacyjny. W przypadku kiedy procesor chce uzyskać dostęp do zmiennej, a wpis w tablicy stron wskazuje, że odpowiadający jej obszar pamięci wirtualnej nie ma przydzielonej ramki w pamięci głównej, zgłaszany jest błąd strony (*page fault*) i uruchamiana procedura jego obsługi. Często błąd strony oznacza tylko konieczność przydzielenia ramki stronie pamięci wirtualnej, bez konieczności odwoływania się do pamięci zewnętrznej, np. kiedy strona nie znajdowała się dotąd w pamięci głównej, a pamięć posiada wolne zasoby. Taki mniejszy błąd strony (*minor page fault*), typowy dla początkowej fazy realizacji programu oraz dla dynamicznej alokacji pamięci, nie stanowi znaczącego narzutu na czas wykonania programu. Bardzo kosztowny jest natomiast większy błąd strony (*major page fault*), związany z przeładowaniem zawartości strony pamięci, pomiędzy pamięcią zewnętrzną a pamięcią główną (*page swap*).

Szczególnie niekorzystnym zjawiskiem przy wykonaniu programu są powtarzające się z dużą częstotliwością żądania dostępu do zbyt wielu stron pamięci, w stosunku do rozmiaru pamięci DRAM, co powoduje nieustanne błędy stron i przeładowania zawartości pomiędzy pamiecią główną i zewnętrzną, określane jako szamotanie (*thrashing*). Szamotanie może być konsekwencją zbyt dużej liczby stron wykorzystywanych bieżąco przez pojedynczy program, może także być związane ze zbyt dużą liczbą uruchomionych programów (z których każdy wymaga częstego dostępu do określonego zbioru stron pamięci wirtualnej).

System operacyjny, w ramach sterowania mechanizmem pamięci wirtualnej, zarządza także tablicą stron. W praktyce współczesne systemy operacyjne stosują wiele tablic stron, np. odrębne dla każdego procesu lub dla większych obszarów pamięci zwanych segmentami (wtedy adres wirtualny zawiera więcej sekcji, np. sekcję dla katalogu tablic stron). Tablice stron przechowywane są w pamięci DRAM, a

także w szybkiej pamięci podręcznej TLB (opisywanej przy omawianiu architektur rdzeni mikroprocesorów, p. 3.5). W skrajnym przypadku bardzo obszerna tablica stron może także, w ramach pamięci wirtualnej, znajdować się częściowo w pamięci zewnętrznej.

4.1.1 Testowanie mechanizmu podmiany stron

Prostym testem stosowania przez system operacyjny podmiany stron (*page swapping*) jest zaalokowanie i specjalne wykorzystywanie w programie tablicy o rozmiarze przekraczającym (najlepiej niewiele) rozmiar pamięci DRAM komputera. Odpowiednie wykorzystanie, powodujące podmiany stron, można uzyskać wielokrotnie realizując dostęp do specjalnie dobranych elementów tablicy (np. oddalonych od siebie o rozmiar strony w pamięci), za pomocą odpowiednio skonstruowanych pętli. Śledzenie podmian stron pamięci wirtualnej można osiągnąć na różne sposoby. Można wykorzystać jedną z procedur systemowych (np. *getrusage* w Linuxie), zwracających dotychczasową liczbę błędów stron w dowolnym momencie wykonania programu. W Linuxie można także wykorzystać narzędzie systemowe *time* (*/usr/bin/time*, w odróżnieniu do polecenia powłoki *time*), które zwraca informacje o błędach stron dla całego programu, przekazanego jako argument polecenia. Podobną informację zwraca, wspominane już, narzędzie *perf stat*. Kolejnym sposobem może być śledzenie aktywności systemu, np. za pomocą polecenia *top* (wymaga to odpowiednio długiego czasu wykonania programu, nawet przy zadaniu częstości próbkowania dla polecenia *top* co wybrany ułamek sekundy, za pomocą np. *top -d 0,1*). Pojawienie się narzutu związanego z podmianą stron widoczne będzie poprzez zmniejszenie procentu czasu CPU przydzielanego wykonywanemu procesowi oraz pojawienie się, ze znacznym udziałem czasu CPU, procesów odpowiedzialnych za obsługę błędu strony (np. demon *kswapd* w Linuxie).

4.2 Pamięć podręczna

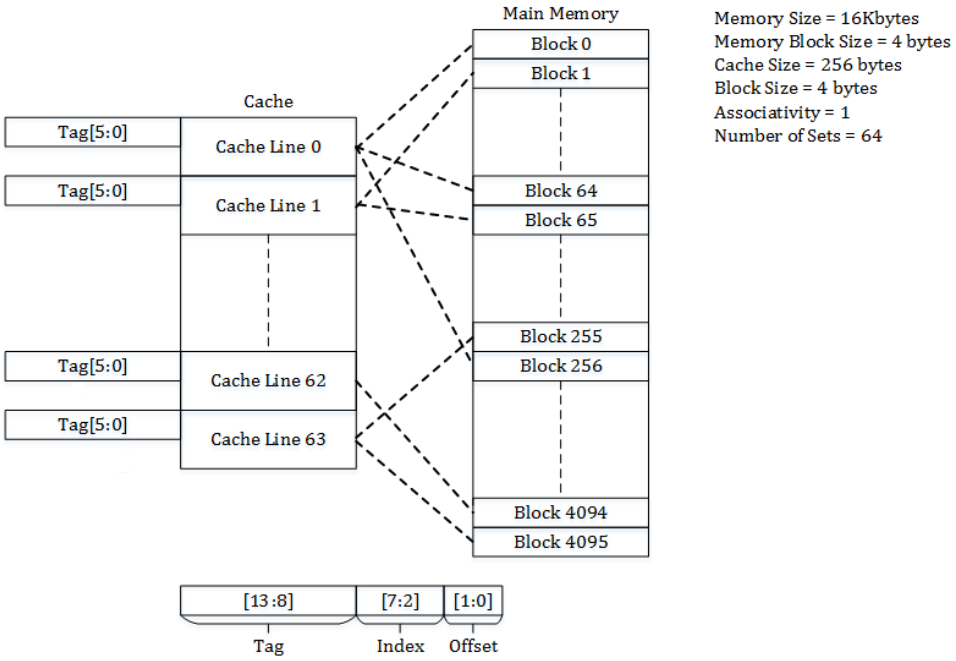
4.2.1 Podstawowy mechanizm działania pamięci podręcznej i lokalność odniesień

Obliczony w ramach mechanizmu pamięci wirtualnej adres fizyczny służy do uzyskania dostępu do zawartości odpowiadającej mu komórki pamięci DRAM[5]. W praktyce dostęp taki jest zawsze realizowany za pośrednictwem pamieci podręcznej, przechowującej kopie danych. Opis mechanizmów funkcjonowania pamięci podręcznej rozpoczniemy od sytuacji występowania tylko pojedynczego jej poziomu, rozszerzając go następnie na przypadek kilku poziomów.

Pamięć podręczna składa się z linii. Dla różnych poziomów pamięci rozmiar pojedynczej linii może się różnić, przy czym najczęściej spotyka się linie o rozmiarze 64 lub 128 bajtów (co oznacza kilka zmiennych podwójnej precyzji lub kilkanaście zmiennych całkowitych i pojedynczej precyzji). Pamięć DRAM podzielona jest na bloki, każdy o rozmiarze odpowiadającym rozmiarowi linii pamięci podręcznej. W najprostszym modelu, pamięci odwzorowanej bezpośrednio (*direct mapped*), każdy blok pamięci DRAM jest przyporządkowany do pojedynczej linii pamięci podręcznej, co oznacza, że kopia zawartości bloku może znajdować się tylko w tej linii. Kolejny następujący po nim blok jest przyporządkowany kolejnej linii, itd. Całkowity rozmiar pamięci podręcznej jest znacznie mniejszy od rozmiaru pamięci DRAM, stąd w pewnym momencie kolejny blok zostaje przyporządkowany pierwszej linii, następujący po nim drugiej itd. W końcowym efekcie, pojedynczej linii pamięci podręcznej odpowiada wiele bloków pamięci DRAM.

Rysunek 4.5 pokazuje zasadę umieszczania danych w pamięci podręcznej dla hipotetycznej sytuacji, kiedy pamięć operacyjna adresowana jest za pomocą 14 bitów (jej rozmiar wynosi więc 16 kilobajtów), a pamięć podręczna składa się z 64 linii o rozmiarze 4 bajtów. Pamięć główna dzielona jest więc na

[5]W praktyce dostęp do pamięci może odbywać się do pewnego stopnia równolegle z translacją adresu z wirtualnego na fizyczny.



Rysunek 4.5: Mechanizm umieszczania zawartości komórek pamięci głównej w liniach pamięci podręcznej i związany z nim sposób wykorzystania kolejnych bitów adresu, dla pamięci odwzorowanej bezpośrednio [źródło: Wikipedia].

4-bajtowe bloki, pierwsze 64 z nich (o numerach od 0 do 63) odwzorowane są w kolejne 64 linie pamięci podręcznej, po czym blok o numerze 64 odwzorowany jest ponownie w pierwszą linię pamięci podręcznej (linia 0), blok o numerze 65 w drugą, kolejne bloki w kolejne linie, aż do bloku o numerze 127, a cały proces powtarza się dla każdego zbioru 64 bloków pamięci. Przy rozmiarze 16 kB, całkowita liczba bloków wynosi 4096, czyli 64 sekwencje kolejnych 64 bloków (w jedną linię pamięci podręcznej odwzorowane są więc 64 bloki pamięci głównej).

Na adres dowolnej jednobajtowej komórki pamięci składają się dwa bity (*offset*), które oznaczają miejsce w czterobajtowym bloku pamięci DRAM, a więc także w czterobajtowej linii pamięci podręcznej, kolejne 6 bitów oznaczających numer bloku w zbiorze 64 kolejnych bloków (czyli indeks odpowiadającej linii pamięci podręcznej – *index*) oraz ostatnie 6 bitów, określających numer zbioru bloków, tworzące etykietę (*tag*). W każdej linii pamięci podręcznej może znajdować się blok z dowolnego 64-blokowego zbioru, etykieta (*tag*) musi być więc przechowywana wraz z linią pamięci podręcznej, aby umożliwić sprawdzenie, który blok pamięci głównej jest aktualnie przechowywany w pamięci podręcznej.

Mechanizm działania pamięci podręcznej jest następujący. Procesor, po wygenerowaniu adresu zmiennej, do której chce uzyskać dostęp, sprawdza czy w pamięci podręcznej znajduje się aktualna kopia wartości zmiennej[6]. W tym celu oblicza na podstawie odpowiednich bitów adresu tworzących indeks (*index*), w której linii może znajdować się kopia bloku pamięci głównej zawierającego zmienną, a następnie na podstawie bitów określających etykietę (*tag*), sprawdza czy rzeczywiście w linii znajduje się ten blok. Jeśli aktualna wartość zmiennej jest w pamięci podręcznej (co będziemy określali jako trafienie w pamięci podręcznej, *cache hit*), procesor realizuje szybki dostęp do zmiennej. Jeśli blok zawierający

[6]Znaczenie określenia aktualna kopia wyjaśnione jest przy omawianiu mechanizmów utrzymania spójności pamięci podręcznej w części drugiej książki, poświęconej obliczeniom wielowątkowym i wieloprocesowym.

zmienną nie znajduje się w pamięci podręcznej (lub jeśli jego zawartość nie jest aktualna), co określane jest jako chybienie (*cache miss*), zawartość całej linii jest podmieniana, odpowiedni blok pamięci DRAM jest kopiowany do linii pamięci podręcznej, po czym następuje ponowienie próby dostępu do zmiennej, tym razem z zapewnionym trafieniem. Załadowany blok pozostaje w pamięci podręcznej, dopóki nie zostanie podmieniony przez inny blok, odwzorowany w tę samą linię.

W oczywisty sposób pierwszy dostęp do każdego bloku pamięci głównej w trakcie działania programu powoduje chybienie w pamięci podręcznej (tzw. **chybienie konieczne**, *compulsory miss*). Jednak kolejne dostępy mogą prowadzić do trafień lub chybień. Opis działania pamięci podręcznej wskazuje na kilka podstawowych z punktu widzenia wydajności faktów:

1. jeśli w trakcie wykonania programu dostęp do pewnej zmiennej następuje wielokrotnie w krótkich odstępach czasu (tzw. **lokalność czasowa**, *temporal locality*), to rośnie prawdopodobieństwo, że kolejne dostępy (poza pierwszym) realizowane będą przy użyciu kopii w pamięci podręcznej, a nie wartości w powolnej pamięci DRAM (warunkiem jest, aby odpowiednia linia pamięci podręcznej zawierająca kopię zmiennej, nie została podmieniona lub zdezaktualizowana pomiędzy kolejnymi dostępami)

2. jeśli w trakcie wykonania programu w krótkim odstępie czasu następują dostępy do różnych zmiennych przechowywanych w tym samym bloku pamięci, odwzorowanym w pojedynczą linię pamięci podręcznej (tzw. **lokalność przestrzenna**, *spatial locality*), to rośnie prawdopodobieństwo, że kolejne dostępy do zmiennych (poza pierwszym dostępem do zmiennej z danego bloku) realizowane będą przy użyciu kopii w pamięci podręcznej, a nie wartości w powolnej pamięci DRAM

3. jeśli następuje chybienie w pamięci podręcznej, to narzut czasowy na obsługę chybienia (*miss penalty*) jest znaczący, ze względu na **konieczność podmiany całej linii pamięci podręcznej**

Skuteczność pamięci podręcznej, jako mechanizmu optymalizacji, zależy od tego jak wiele programów i w jak dużym stopniu wykazuje czasową i przestrzenną lokalność dostępów (odniesień) do pamięci (*locality of reference*). Okazuje się, że zdecydowana większość programów w naturalny sposób posiada te dwie cechy i w praktyce korzysta z istnienia pamięci podręcznych, uzyskując dzięki nim redukcję czasu wykonania.

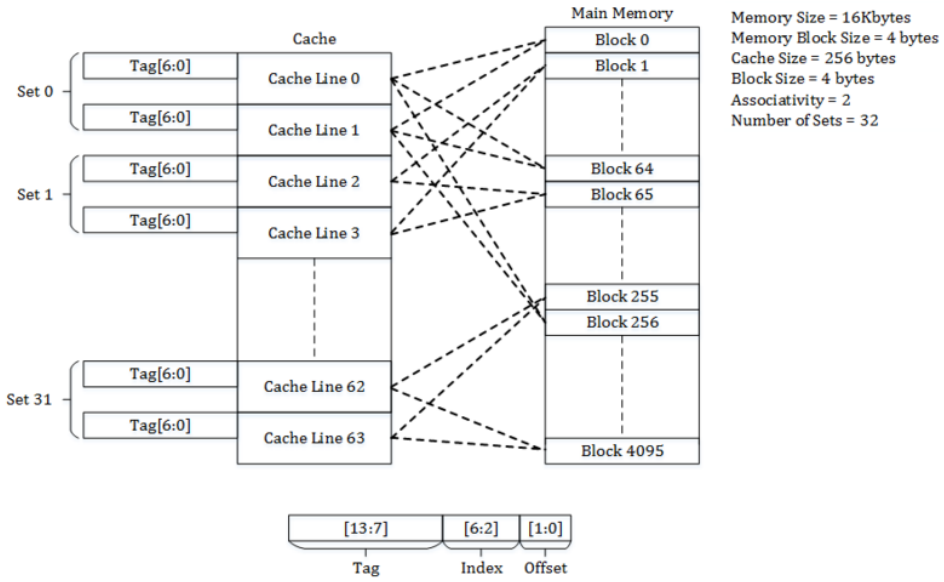
W praktyce wykorzystuje się szereg dodatkowych szczegółowych mechanizmów funkcjonowania pamięci podręcznych, z których niektóre są omówione w dalszej części rozdziału. Już jednak podstawowy sposób działania pamięci podręcznej implikuje najważniejsze wskazanie dotyczące optymalizacji dostępów do pamięci przy wykonaniu programów na współczesnych systemach komputerowych: **należy zawsze dążyć do maksymalizacji lokalności czasowej i przestrzennej odniesień do pamięci w trakcie wykonania programów**. Należy także zawsze pamiętać o tym, że **dostęp do pamięci DRAM (pobranie lub zapis danych) standardowo dotyczy całej linii pamięci podręcznej, a nie pojedynczej zmiennej**.

4.2.2 Drożność pamięci podręcznej

Mechanizm bezpośrednio odwzorowanej pamięci podręcznej przedstawiony dotychczas ma istotną wadę. Rozważmy prostą pętlę algorytmu obliczania iloczynu skalarnego dwóch wektorów o N elementach:

```
for (j=0; j<N; j++) il_skal += b[j] * c[j];
```

Wydaje się, że algorytm będzie realizowany optymalnie, ze względu na dużą lokalność przestrzenną odniesień (skok równy 1 dla każdego z wektorów). Jednak może się zdarzyć, że początki tablic `b` i `c`



Rysunek 4.6: Mechanizm umieszczania zawartości komórek pamięci głównej w liniach pamięci podręcznej i związany z nim sposób wykorzystania kolejnych bitów adresu, dla dwudrożnej pamięci sekcyjno-skojarzeniowej [źródło: Wikipedia].

zostaną odwzorowane w to samo miejsce pamięci podręcznej (dla rys. 4.5 mogłoby się tak zdarzyć, gdyby np. tablica `b` zaczynała się w bloku 0, a tablica `c` w bloku 256). Wtedy w pierwszej iteracji, po pobraniu elementu tablicy `b`, przy próbie pobrania elementu tablicy `c` następuje chybienie w pamięci podręcznej i kosztowna czasowo podmiana całej linii. W drugiej iteracji, przy próbie dostępu do drugiego elementu tablicy `b`, ponownie następuje chybienie, mimo że element ten przed chwilą był już w pamięci podręcznej, jednak jego blok został podmieniony przez blok związaną z tablicą `c`. W tej samej drugiej iteracji następuje w chwilę później kolejne chybienie, przy dostępie do drugiego elementu tablicy `c`, który znowu przed chwilą był w pamięci podręcznej, lecz jego blok został podmieniony elementami tablicy `b`. Dzieje się tak dla każdej kolejnej iteracji, więc w efekcie w każdej iteracji pętli występują dwa chybienia w pamięci podręcznej i dwie podmiany linii.

Aby temu zapobiec powszechnie stosuje się tzw. pamięci sekcyjno-skojarzeniowe (*set associative*). Pojedynczy blok pamięci DRAM jest przyporządkowany więcej niż jednej linii pamięci podręcznej, w praktyce najczęściej 2, 4 lub 8 liniom, co nazywane jest pamięcią dwudrożną (*two-way*), czterodrożną (*four-way*) i ośmiodrożną (*eight-way*).

Rysunek 4.6 pokazuje schemat działania pomięci dwudrożnej. Cała pamięć podręczna dzielona jest na zbiory (*set*) po dwie linie w zbiorze (dla pamięci czterodrożnej byłyby to zbiory po cztery linie, dla ośmiodrożnej po osiem linii, itd). Każdy blok w pamięci głównej (o rozmiarze pojedynczej linii pamięci podręcznej) może być przechowywany w jednej z dwóch linii zbioru, w który został odwzorowany. Na rys.4.6 oznacza to np. że blok 0 pamięci głównej może być przechowywany w linii 0 lub linii 1, stanowiących zbiór 0. Z kolei linie 2 i 3 tworzą zbiór 1, w który odwzorowany jest blok 1 pamięci głównej. W przykładowej zilustrowanej pamięci podręcznej znajdują się 32 zbiory (dla 64 linii), co oznacza, że blok 31 pamięci głównej odwzorowany jest w zbiór 31, a blok 32 (podobnie jak bloki 64, 96, 128 itd.) w zbiór 0.

Zmiana sposobu odwzorowania bloków pamięci DRAM w linie pamięci podręcznej powoduje, że teraz część adresu stanowiąca indeks (*index*) obejmuje tylko 5 bitów (odpowiadając liczbie zbiorów linii), natomiast część stanowiąca etykietę (*tag*) ma 7 bitów - w pojedynczy zbiór może być odwzorowane

dwa razy więcej bloków niż w pojedynczą linię pamięci podręcznej bezpośrednio odwzorowanej. Sprawdzenie czy konkretna linia pamięci podręcznej zawiera żądany blok pamięci DRAM obejmuje teraz, dla pojedynczej wartości indeksu, dwie linie (dwa razy więcej niż poprzednio) oraz 7 bitów etykiety dla każdej linii, w miejsce 6.

Dla przykładowego algorytmu mnożenia skalarnego pamięć dwudrożna usuwa niebezpieczeństwo ciągłych podmian linii w pamięci podręcznej, które groziły w przypadku pamięci bezpośrednio odwzorowanej. Nawet jeśli początek tablicy `b` odwzorowany będzie w zbiór 0, tak samo jak początek tablicy `c`, to poprawnie działający mechanizm podmiany linii w pamięci podręcznej, umieści w trakcie wykonania algorytmu iloczynu skalarnego początkowe wyrazy tablicy `b` w jednej linii zbioru, a początkowe wyrazy tablicy `c` w drugiej.

W kolejnych iteracjach, bloki pamięci głównej zawierające kolejne fragmenty tablic będą umieszczane w kolejnych zbiorach linii pamięci podręcznej. Linie w zbiorach będą wypełniane danymi, na których algorytm dokonuje operacji, aż do momentu dojścia do ostatniego zbioru. W takiej sytuacji wszystkie linie będą wypełnione danymi z tablic. Jeśli rozmiary tablic będą odpowiednio duże, przejście do kolejnej iteracji i do kolejnych elementów tablic, wymusi podmianę linii w pamięci podręcznej. Przyczyną tej podmiany będzie fakt, że rozmiar danych algorytmu przekroczył rozmiar pamięci podręcznej. Podmiana w takiej sytuacji następuje w wyniku chybienia w pamięci podręcznej, wymuszonego przez zbyt małą jej pojemność. Mówimy wtedy o **chybieniu pojemnościowym** (*capacity miss*).

Pamięć dwudrożna rozwiązuje problem możliwych częstych chybień w przypadku algorytmu iloczynu skalarnego, jednak nie usuwa zagrożenia w przypadku np. poniższego dodawania wektorów:

```
for (j=0; j<N; j++) a[j] += b[j] + c[j];
```

Jeśli teraz początki wszystkich trzech tablic odwzorowane zostaną w ten sam zbiór dwóch linii, w każdej iteracji będzie dochodziło do chybień w pamięci podręcznej. Już w pierwszej iteracji po pobraniu z pamięci DRAM początkowych wyrazów dwóch tablic – do dwóch linii w pojedynczym zbiorze, próba pobrania wyrazów trzeciej tablicy doprowadzi do konieczności podmiany jednej z linii zbioru. Stanie się tak, mimo że większość pamięci podręcznej jest w tym momencie niewykorzystywana przez algorytm, a więc nie jest przekraczana pojemność pamięci przez dane algorytmu. Taki typ chybienia w pamięci podręcznej, powodującego konieczność podmiany linii na skutek odwzorowania wielu bloków pamięci DRAM w ten sam zbiór linii pamięci podręcznej, nazywany jest chybieniem na skutek konfliktu (*conflict miss*). Takie **chybienia konfliktowe** są niekorzystne, powodują konieczność podmiany linii mimo pozostawania części pamięci podręcznej niewykorzystanej, utrudniają także analizę i optymalizację algorytmów, łatwiejszą w przypadku chybień pojemnościowych.

W przypadku pamięci odwzorowanej bezpośrednio, ryzyko chybień konfliktowych jest relatywnie duże. Maleje ono w przypadku pamięci dwudrożnej, a dalsze zwiększanie drożności prowadzi do coraz mniejszego niebezpieczeństwa wystąpienia chybień konfliktowych. Im wyższa drożność tym mniejsze prawdopodobieństwo, że w konkretnym programie dla pewnego zbioru zmiennych, intensywnie wykorzystywanych i przyporządkowanych temu samemu zbiorowi linii pamięci podręcznej, liczba linii w zbiorze okaże się zbyt mała, co będzie prowadzić do częstych chybień konfliktowych i podmian linii, podczas gdy pozostałe zbiory linii nie będą w pełni wykorzystane.

Całkowita eliminacja chybień konfliktowych następuje w przypadku tzw. pamięci w pełni skojarzeniowej, *fully associative*, dla której każdy blok pamięci może być przechowywany w dowolnej linii pamięci podręcznej. Daje to gwarancję, że zawsze wykorzystywana będzie pełna pojemność pamięci, nie doprowadzając do marnowania zasobów.

Jednak w praktyce tego typu pamięci stosuje się rzadko, zazwyczaj drożność pamięci podręcznej jest ograniczona. Wynika to z badań statystycznych zysku, jaki przynosi zwiększenie drożności dla wydajności dostępów do pamięci. Zwiększając drożność, zmniejsza się częstotliwość chybień konfliktowych

i przez to zmniejsza narzut związany z obsługą chybień na czas dostępu do danych. Jednak jednocześnie zwiększa się koszt obsługi pamięci podręcznej, i to dla każdego dostępu do pamięci, nie tylko w przypadku rzadkich chybień konfliktowych. Na skutek opisanego wyżej mechanizmu sprawdzania czy występuje trafienie czy chybienie w pamięci podręcznej, zwiększanie drożności pamięci prowadzi do rosnącego narzutu czasowego, a także większego wydatku energetycznego (konieczność każdorazowego sprawdzenia większej liczby linii w pojedynczym zbiorze linii pamięci oraz większa liczba bitów tworzących etykietę w każdym adresie). Dla pewnej wartości drożności, narzuty te zaczynają przeważać nad zyskiem w postaci redukcji liczby chybień konfliktowych.

Fakt ograniczenia drożności pamięci podręcznych ma znaczenie dla optymalizacji wydajności, w szczególności dla algorytmów operujących na tablicach (wektorach, macierzach). Z jednej strony, w przypadku wielu tablic należy sprawdzać czy ich położenie nie prowadzi do chybień konfliktowych. Z drugiej strony, chybienia konfliktowe możliwe są także wtedy, gdy algorytm operuje na małej liczbie tablic, a nawet na pojedynczej tablicy. Dzieje się tak, kiedy dostęp do tablic realizowany jest nie wyraz po wyrazie, ale z pewnym skokiem. Przy pewnych wartościach skoku niebezpieczeństwo chybień konfliktowych rośnie.

4.2.3 Dalsze szczegóły funkcjonowania pamięci podręcznej

Najważniejszym aspektem, dotąd nieporuszanym, funkcjonowania pamięci podręcznej jest występowanie wielu jej poziomów, w dalszej części ograniczonych do L1, L2 i L3. Pamięci różnych poziomów mogą występować jako pamięci zawierające się w sobie (*inclusive caches*) i pamięci odrębne (*exclusive caches*). W pierwszym przypadku, jeśli zawartość bloku pamięci DRAM znajduje się w pamięci bliższej potokom (np. L1), to ten sam blok znajduje się w pamięci kolejnego poziomu (np. L2). Oznacza to, że cała zawartość pamięci bliższej potokom, jest przechowywana także w kolejnej pamięci (czyli w efekcie np. L1 jest ”zawarta” w L2). Pobranie bloku z pamięci DRAM powoduje kolejno wypełnianie linii w L3, L2 i L1.

Inaczej jest dla pamięci odrębnych, których przykładem jest pamięć *victim cache*, służąca do przechowywania podmienionych linii z pamięci bliższej potokom wykonania. Jeśli rolę *victim cache* pełni np. L2, wtedy pobranie z L3 dokonywane jest bezpośrednio do L1, a podmiana w L2 następuję w efekcie usunięcia linii z L1 i jej zapisu do L2. Zawartości L1 i L2 są różne – dostępna pojemność szybkiej pamięci podręcznej tym samym rośnie (jest sumą pojemności L1 i L2, a nie wyłącznie pojemnością L2), choć wzrasta także złożoność mechanizmu dostępu.

Analiza wykorzystania pamięci podręcznej komplikuje się w dalszym stopniu, kiedy uwzględni się fakt, że pamięci poziomów L1 i L2 są najczęściej we współczesnych procesorach ogólnego przeznaczenia umieszczane jako prywatne wewnątrz każdego rdzenia, natomiast poziom L3 jest wspólny dla wielu rdzeni. W przypadku obliczeń wielowątkowych zastosowanie strategii pamięci odrębnej dla pamięci L3 prowadzi do dalszych trudności przy analizie i optymalizacji jej użycia.

Innym, istotnym aspektem funkcjonowania pamięci wielodrożnych, jest fakt, że podmiana zawartości w pamięci, związana z pobraniem nowego bloku, może dotyczyć teoretycznie dowolnej linii ze zbioru linii, w które odwzorowany jest ten blok. Istnieją różne strategie podmiany linii, niektóre faworyzujące szybkość działania nad maksymalizację liczby trafień, jak np. podmiana losowa lub wykorzystanie prostej kolejki FIFO, inne bardziej złożone, jak np. podmiana linii najdawniej użytej (*LRU, least recently used*) lub najrzadziej używanej (*LFU, least frequently used*). W praktyce wykorzystywane bywają strategie jeszcze bardziej złożone, co prowadzi do trudności jednoznacznego ustalenia oddziaływania konkretnych rozwiązań z kodu źródłowego na funkcjonowanie podmian. W efekcie funkcjonowanie podmian nie jest analizowane w niniejszej książce w swojej ogólności, czasem jednak uwzględniany będzie jego możliwy wpływ na wydajność konkretnych programów.

4.3 Praktyczne aspekty wykorzystania hierarchii pamięci

Realizacja dostępów do pamięci we współczesnych systemach komputerowych obejmuje, poza omówionymi powyżej zasadami funkcjonowania pamięci podręcznej i wirtualnej, cały szereg innych elementów sprzętowych i mechanizmów ich działania.

Mechanizmami takimi są m.in. szczegóły technologii SRAM i DRAM, a także funkcjonowanie ewentualnych dodatkowych układów stosowanych przez producentów procesorów w celu usprawnienia dostępów do pamięci (jak np. działanie układów udostępniania elementów podmienianej linii w pamięci podręcznej, jeszcze przed zakończeniem pobierania całej linii, wielobankowość, wielokanałowość pamięci itp.). Wiele z tych mechanizmów ma na celu zwiększenie możliwego stopnia współbieżności dostępów do pamięci i dzięki temu ukrywanie opóźnienia związanego z izolowanym dostępem.

Mimo że omówione dotychczas mechanizmy funkcjonowania pamięci mają znaczący wpływ na wydajność wykonania programów, jednak do ich wykorzystania nie ma zazwyczaj standardowych jawnych technik programowania. Już sam fakt istnienia pamięci podręcznej nie jest zaznaczony w praktycznie żadnym z najważniejszych języków programowania. Optymalizacja użycia pamięci podręcznej odbywa się poprzez zmiany kodu źródłowego, co do których przewiduje się, że będą miały wpływ na zwiększenie wydajności obliczeń, jednak takie heurystyczne podejście nie gwarantuje pewnego rezultatu i bywa nieprzenośne[7].

W niniejszym punkcie, przed przystąpieniem do badań eksperymentalnych dotyczących dostępów do pamięci, omówione są dwa aspekty funkcjonowania pamięci, mające wpływ na wydajność i mogące znaleźć odzwierciedlenie na poziomie kodu źródłowego programów. Pierwszym z nich jest wyrównanie zmiennych w pamięci, tak aby rozpoczynały się w komórkach o określonych adresach, należących do określonego podzbioru całego dostępnego zbioru adresów (*alignment*), a drugim pobieranie z wyprzedzeniem (*prefetching*), realizowany sprzętowo, ale także mogący być wprowadzony jako optymalizacja kodu.

4.3.1 Położenie zmiennych w pamięci głównej

Charakterystyki współczesnego sprzętu komputerowego (rozmiary rejestrów, szerokości magistral danych, sposób działania kości DRAM, itp.), powodują, że maksymalna wydajność układu pamięci osiągana jest dla wielobajtowych zestawów danych. Dotyczy to głównie tablic i innych złożonych struktur danych, ale optymalizacji podlega także dostęp do pojedynczych zmiennych podstawowych typów danych, o rozmiarach kilku bajtów.

Standardowo, kompilatory działają tak, aby pojedyncze zmienne konkretnego typu (mające najczęściej rozmiary 4 lub 8 bajtów) miały adresy swoich początków będące wielokrotnością swojego rozmiaru. Mówimy wtedy o wyrównaniu (*data alignment*) na granicy 4- lub 8-bajtowej[8].

W przypadku tablic wpływ na wydajność ma położenie nie tylko pojedynczych zmiennych, ale także sekwencji kolejnych wyrazów. Pojawia się wtedy problem odpowiedniego wykorzystania nie tylko wymienionych wcześniej elementów sprzętowych (rejestry, magistrale i moduły pamięci DRAM), ale także pamięci podręcznej. Ze względu na rozmiar linii pamięci podręcznych będący wielokrotnością 8 bajtów, przy zmiennych wyrównanych standardowo w pamięci nie zachodzi niebezpieczeństwo odwzorowania jednej zmiennej częściowo do jednej, a częściowo do kolejnej linii pamięci podręcznej. Dla krótkich

[7] Alternatywą jest korzystanie z wybranych rozkazów z listy procesora operujących jawnie na pamięci podręcznej, które jednak także nie dają możliwości sterowania większością aspektów jej funkcjonowania.

[8] Oznacza to użycie wybranego podzbioru możliwych adresów, przy założonym adresowaniu pojedynczych bajtów pamięci. Możliwą konsekwencją takiej praktyki jest występowanie niewypełnionych danymi przestrzeni wewnątrz złożonych struktur danych, np. struktur języka C. Jeśli struktura jest wyrównana na granicy 4-bajtowej, jej pierwszym elementem jest pojedyncza zmienna znakowa, a drugim zmienna np. całkowita, to przy wyrównaniu zmiennej całkowitej na granicy 4-bajtowej, pomiędzy pierwszym i drugim elementem struktury pozostaną trzy bajty niezapełnione danymi programu.

tablic lub dla algorytmów, które operują na fragmentach tablic, znaczenie może mieć jak sekwencja elementów odwzorowana jest w zbiór linii pamięci podręcznej.

Najczęściej zakłada się, że optymalne ułożenie w pamięci oznacza początek tablicy pokrywający się z początkiem bloku odwzorowanego w pojedynczą linię pamięci podręcznej. Aby to uzyskać wystarczy, aby początkowy adres tablicy był wielokrotnością rozmiaru linii pamięci podręcznej. W trakcie wykonania programu, pierwszy wyraz tablicy znajduje się wtedy na pewno na początku pewnej linii pamięci podręcznej. W praktyce stosuje się w takim przypadku wyrównanie na granicy 64-bajtowej (rzadziej 128-bajtowej).

Do alokowania w pamięci dynamicznej tablic wyrównanych na odpowiedniej granicy służą rozmaite narzędzia, najczęściej nie będące składową języka programowania. W przypadku wielu kompilatorów języka C można posłużyć się przenośną funkcją standardu POSIX:

```
int posix_memalign(void **memptr, size_t alignment, size_t size);
```

Jedyną istotną różnicą w stosunku do standardowej alokacji w języku C (poza innym sposobem zwracania wskaźnika `memptr` do zaalokowanej pamięci) jest podanie, oprócz rozmiaru alokowanej pamięci `size` (w bajtach), także żądanego wyrównania `alignment`, ponownie wyrażanego w bajtach.

W niniejszej książce, przy opisach działania sprzętu i wykonania rozmaitych algorytmów, a także w programach związanych z książką, standardowo zakładane jest wyrównanie wszystkich tablic na granicy 64-bajtowej.

4.3.2 Pobieranie z wyprzedzeniem przy realizacji dostępów do pamięci

Wspominany już uprzednio, jako przykład techniki wykonania spekulatywnego, mechanizm sprzętowy pobierania z wyprzedzeniem (*hardware prefetching*) w istotny sposób wpływa na funkcjonowanie dostępów do pamięci we współczesnych procesorach (stosują go praktycznie wszystkie mikroprocesory, czasem w sposób zaawansowany, np. z kilkoma układami współbieżnie realizującymi tego typu pobieranie). Ma on między innymi znaczenie przy wszelkich próbach wykorzystania mikrobenchmarków do pomiaru wydajności pamięci, a także tworzenia modeli wydajnościowych wykonania programów. Poniżej omówiony jest prosty przykład algorytmu, często stosowanego przy badaniu parametrów pamięci podręcznej, pokazujący podstawowe zasady funkcjonowania pobierania z wyprzedzeniem oraz ich konsekwencje dla wydajności wykonania.

Analizowanym algorytmem jest wykonanie prostej pętli, w której sumuje się wartości w tablicy liczbowej `tab` o rozmiarze `rozmiar_tab`:

```
for (j=0; j<rozmiar_tab; j++) suma += tab[j];
```

Zgodnie z przyjętą konwencją, założeniem badania jest wyrównanie tablicy na granicy będącej wielokrotnością rozmiaru linii pamięci podręcznej.

Kod asemblera dla badanej pętli może wyglądać następująco (poniższy kod utworzony został przez kompilator *gcc*):

```
.L5:
        addsd   0(%rbp,%rdx), %xmm7
        addq    $8, %rdx
        cmpq    %rax, %rdx
        jne     .L5
```

Nie wchodząc w możliwe warianty kodu (obejmujące użycie innych rozkazów procesora, rozwinięcie pętli przez kompilator, czy zastosowanie innych typów rejestrów), podstawowe działania w każdej iteracji pętli przebiegają w ten sam sposób: realizowany jest dostęp do pamięci przy wykonaniu rozkazu

dodania zawartości elementu tablicy do sumy (przechowywanej w rejestrze `%xmm7`), następnie wykonywane są operacje na zawartości rejestru `%rdx`, związanego z indeksem pętli i adresem elementu tablicy, po czym sterowanie, zależnie od wyniku porównania zawartości `%rdx` z graniczną wartością oznaczającą koniec pętli przechowywaną w rejestrze `%rax`, albo powraca na początek pętli, albo przechodzi do dalszej części kodu[9].

Naiwne działanie procesora przy realizacji kodu w każdej kolejnej iteracji mogłoby polegać na wykonaniu wszystkich rozkazów danej iteracji i dopiero po ich ukończeniu przejście do następnej iteracji. Czas realizacji tak wykonywanego kodu byłby sumą czasów realizacji poszczególnych iteracji, z rozkazami dostępu do pamięci w każdej z nich.

Zgodnie z omawianymi wcześniej zasadami funkcjonowania potoków przetwarzania, zgodnie z którymi układ wykonywania rozkazów dąży do maksymalizacji współbieżności działania, nie tylko każdy potok stara się indywidualnie realizować współbieżnie wiele rozkazów, ale także, dzięki liczbie i różnorodności potoków, możliwe staje się całkowicie równoległe realizowane kilku rozkazów różnych typów. W budowie współczesnych rdzeni, jak np. rdzenie prezentowane w p. 3.7, ważną cechą jest istnienie kilku niezależnych potoków przeznaczonych do wykonywania rozkazów dostępów do pamięci (rozkazy *load*, *store*, *mov*). Przy optymalizacji wykorzystania tych potoków (a także powiązanych innych elementów sprzętowych), podobnie jak dla potoków wykonywania operacji arytmetycznych, kluczowym czynnikiem, zgodnie z prawem Little'a, będzie zdolność zapełnienia potoków wystarczającą liczbą niezależnych rozkazów pobierania i zapisu z i do pamięci.

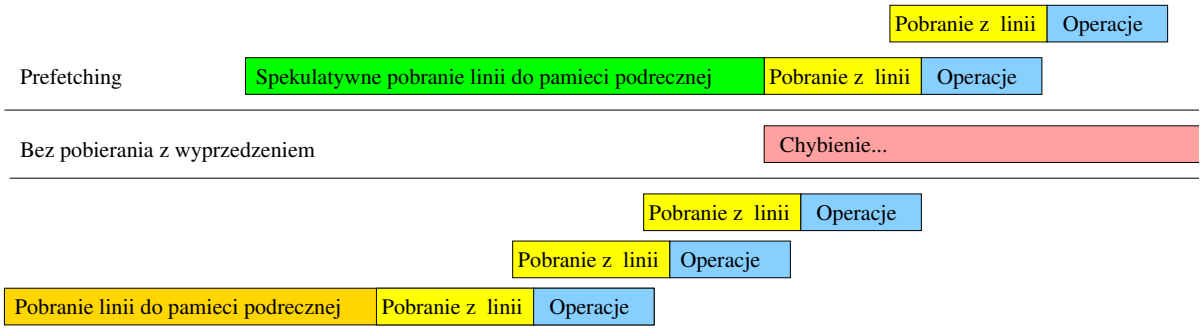
Analizując wykonanie przykładowej omawianej pętli, można zaobserwować, że procesor wykorzystując swoje możliwości współbieżnego działania (nie tylko użycie wielu potoków przetwarzania, ale także przewidywanie skoków czy wykonywanie poza kolejnością) jest w stanie kolejno wygenerować zestaw żądań dostępu do pamięci dla kolejnych wyrazów tablicy. Zakładając, że czas obsługi takiego żądania (nawet realizowanego z pamięci podręcznej) jest dłuższy niż czas wykonywania pozostałych operacji w pojedynczej iteracji pętli, można założyć, że wszystkie te pozostałe operacje zostaną wykonane współbieżnie z dostępem do pamięci, niejako w jego tle, i że czas wykonania pętli będzie wyłącznie czasem realizacji dostępów do pamięci. Takie założenie ukrywania czasów realizacji wszelkich innych rozkazów w czasie realizacji rozkazów dostępu do pamięci jest wykorzystywane we wszystkich analizach wydajności pamięci w niniejszym rozdziale.

Schematycznie i przykładowo możliwy sposób działania procesora i układu pamięci przy realizacji pętli przedstawia rys. 4.7. Wzdłuż kierunku poziomego na rysunku biegnie czas, natomiast kolejne, licząc od dołu, prostokątne paski odpowiadają kolejnym iteracjom pętli. Dla prostoty w dalszej analizie założony jest tylko jeden poziom pamięci podręcznej, co jednak łatwo daje się uogólnić na wielopoziomową pamięć podręczną.

W części rysunku poniżej poziomej linii, założona jest omawiana wyżej współbieżność działania prowadząca do sytuacji, kiedy czas wykonania jest w całości determinowany przez czas operacji na hierarchii pamięci. Po początkowym pobraniu z pamięci DRAM pojedynczej linii pamięci podręcznej, w kilku kolejnych iteracjach wykorzystywane są dane z pobranej linii. Generowane przez procesor żądania dostępu do pamięci skutkują trafieniami w pamięci podręcznej co powoduje szybszy transfer danych do procesora.

Po kilku trafieniach, z konieczności pojawia się chybienie i długotrwała procedura podmiany linii w pamięci podręcznej. Wykonywanie iteracji jest wstrzymywane do czasu dostarczenia nowych danych do

[9] W praktyce należy spodziewać się, że optymalizujący kompilator przy tworzeniu kodu asemblera dokona rozwinięcia pętli (optymalizacja rozwijania pętli (*loop unrolling*) omawiana jest w p. 5.1). W takim przypadku pierwszy rozkaz iteracji (pobranie danych i dodawanie) wykonywany jest wielokrotnie dla każdej pojedynczej operacji na indeksie pętli i skoku na początek pętli. Ma to istotne znaczenie nie tylko dla redukcji liczby rozkazów wykonywanych w całej pętli, ale także dla możliwości współbieżnego działania sprzętu. Dalsza analiza wykonania posługuje się założeniem kolejnego wykonywania pojedynczych iteracji, pozostaje jednak słuszna także po rozwinięciu pętli.



Rysunek 4.7: Schemat mechanizmu pobierania z wyprzedzeniem (*prefetching*)

linii pamięci podręcznej – pokazuje to na rysunku schemat funkcjonowania bez pobierania z wyprzedzeniem. Całkowity czas wykonania pętli jest sumą czasów obsługi trafień i chybień w pamięci podręcznej, a więc czasów dostępów do pamięci podręcznej i pamięci DRAM, zakładając pewien stopień współbieżności działania.

Powyżej tego schematu znajduje się ilustracja możliwego działania mechanizmu *prefetchingu*. Stara się on zoptymalizować działanie układu procesor-pamięć poprzez zainicjowanie transferu danych z pamięci DRAM jeszcze przed wystąpieniem chybienia w pamięci podręcznej, natychmiast kiedy tylko dostępne są wymagane elementy sprzętowe. Na podstawie dotychczasowych dostępów do pamięci, odpowiednie układy spekulatywnie przewidują, gdzie nastąpi kolejny dostęp i uruchamiają procedury pobierania dla odpowiedniego bloku pamięci DRAM. W efekcie nie tylko operacje arytmetyczne i logiczne, ale także pobrania z pamięci podręcznej mogą przebiegać w tle pobrań z pamięci DRAM. Czas wykonania programu staje się w pełni determinowany wyłącznie przez współbieżną realizację żądań dostępu do pamięci głównej.

Wczesne generowanie żądań dostępu do pamięci (tak jak to pokazuje rys. 4.7) nie tylko pozwala na ukrycie w tle czasów realizacji innych operacji, ale także umożliwia zwiększenie wydajności dostępów do pamięci. Układ realizacji operacji na hierarchii pamięci, posiadający znaczące opóźnienie wykonywania rozkazów, pozwala je ukryć, dzięki potokowej, nieblokującej realizacji wielu transferów danych. Pobieranie z wyprzedzeniem jest jednym z mechanizmów maksymalizacji liczby współbieżnie generowanych żądań dostępu do pamięci, co, zgodnie z prawem Little'a, powinno zmierzać do optymalnego wykorzystania możliwości sprzętu.

Pobieranie z wyprzedzeniem jest przykładem wykonania spekulatywnego, kiedy procesor lub inny układ sprzętowy realizuje rozkazy i operacje, co do których nie wiadomo czy rzeczywiście będą wymagane w programie[10]. Wykonanie spekulatywne wymaga istnienia mechanizmów, które powodują poprawne wykonanie programu, kiedy przewidywania okazują się błędne. W przypadku *prefetchingu* jest to relatywnie proste i polega na pominięciu niepotrzebnie pobranych danych.

Pobieranie z wyprzedzeniem zazwyczaj znacząco zwiększa wydajność programów, w szczególności w przypadku intensywnego korzystania z pamięci. Skuteczność pobierania z wyprzedzeniem zależy od zdolności poprawnego przewidzenia, do których bloków pamięci odnosić się będą kolejne żądania dostępu w trakcie wykonania. W praktyce skuteczność wynika z faktu, że dostępy do pamięci są często realizowane przez programy według pewnego wzorca. W przypadku analizowanej powyżej pętli wzorcem tym jest dostęp do kolejnych komórek pamięci oddalonych o stały, ściśle określony odstęp.

Współczesne procesory mogą posiadać kilka układów pobierania z wyprzedzeniem, dotyczących np. transferów pomiędzy różnymi poziomami pamięci podręcznej i pamięcią DRAM. Układy te można wyłączać z poziomu systemu operacyjnego, jeśli np. chce się uzyskać informację o pracy konkretnego

[10]Przykładem takiego mechanizmu wykonania spekulatywnego było także przewidywanie skoków, omawiane w p. 3.4.4.

komponentu sprzętowego, a pobieranie z wyprzedzeniem zaburza uzyskiwanie takiej informacji. W przykładach w niniejszej książce wyłączanie pobierania z wyprzedzeniem nie jest stosowane, natomiast uwzględniany i badany będzie możliwy wpływ *prefetchingu* na wydajność realizacji analizowanych algorytmów. Odpowiada to punktowi widzenia przeciętnego użytkownika systemów komputerowych, który przy wykonywaniu programów nie ma możliwości uniknięcia efektów pobierania z wyprzedzeniem.

4.4 Eksperymentalne określanie podstawowych charakterystyk pamięci głównej i pamięci podręcznych różnych poziomów

Mikrobenchmarkami wykorzystywanymi w niniejszym punkcie są wykonania różnych wariantów prostej pętli (zbliżonej w swym charakterze do pętli użytej przy analizie *prefetchingu*), w której odwiedzane są po kolei wyrazy pewnej tablicy liczbowej `tab` o rozmiarze `rozmiar_tab`, oddalone od siebie o `skok` elementów:

```
for (j=0; j<rozmiar_tab; j+= skok) ... tab[j] ...;
```

Na wyrazach tablicy wykonywane są proste operacje, o czasach realizacji znacząco krótszych od czasu pobrania pojedynczej danej z dowolnego poziomu pamięci podręcznej. Tablica jest wyrównana na granicy odpowiadającej długości linii pamięci podręcznej. Analiza pracy mikroprocesora i układu pamięci, analogiczna jak dla przykładu z poprzedniego punktu, obejmująca m. in. mechanizmy pobierania z wyprzedzeniem i potokowego przetwarzania rozkazów dostępu do pamięci, wskazuje, że w przypadku każdego z mikrobenchmarków jedynym składnikiem istotnym dla całkowitego czasu wykonania jest czas operacji na hierarchii pamięci. Mimo swojej prostoty kod pozwala na wykrycie różnic w czasie wykonania dla rozmaitych kombinacji wartości poszczególnych parametrów pętli i wnioskowanie, na podstawie tych różnic, o wybranych aspektach mechanizmów funkcjonowania hierarchii pamięci oraz ich wpływie na wydajność.

4.4.1 Eksperymentalne wykrywanie rozmiarów pamięci podręcznych różnych poziomów

W celu ustalenia rozmiaru pamięci podręcznych kolejnych poziomów wykorzystywana jest następująca wersja przedstawionej powyżej pętli:

```
for (j=0; j<rozmiar_tab; j++) tab[j]++;
```

Badana pętla wykonywana jest dla różnych rozmiarów `rozmiar_tab` tablicy `tab`, z wielokrotnym powtórzeniem dla każdego z rozmiarów (wielokrotne wykonanie dla każdego z rozmiarów jest dodatkowo powtarzane kilka razy w celu uzyskania większej dokładności pomiaru czasu oraz dla uniknięcia efektu zaburzeń przy pierwszym wykonaniu).

W badaniu tym, skok pomiędzy elementami tablicy modyfikowanymi w kolejnych iteracjach pętli wynosi 1. Jest to optymalna wartość, najczęściej pojawiająca się w standardowych algorytmach, dla której i kompilator, i sprzęt mogą zastosować szereg technik i mechanizmów optymalizacji. Przykładowo kompilator może zastosować rozkazy wektorowe, a sprzęt pobieranie z wyprzedzeniem.

W trakcie wykonania pętli, pobierane z pamięci i modyfikowane są następujące po sobie wyrazy tablicy. Występuje lokalność przestrzenna, kolejne wyrazy znajdują się w pamięci bezpośrednio po sobie. Po chybieniu w pamięci podręcznej związanym z dostępem do pierwszego elementu w bloku pamięci DRAM i pobraniu bloku do linii pamięci podręcznej, dostęp do kilku następnych elementów (ilu to zależy od rozmiaru linii i rozmiaru danych) powoduje trafienie i szybki transfer z pamięci podręcznej.

W kolejnych iteracjach wyrazy wypełniają linie pamięci podręcznej w sposób gęsty, każda linia jest wypełniana zawartością bloku pamięci, każdy pobrany z pamięci operacyjnej wyraz jest wykorzystywany w algorytmie. Przepustowość magistrali i układów DRAM nie jest marnowana na przesyłanie danych niewykorzystywanych w obliczeniach, pracują one (między innymi także dzięki pobieraniu z wyprzedzeniem) z pełną wydajnością.

Jeżeli cała tablica mieści się w pamięci podręcznej, to kolejne wykonanie pętli będzie pracować tylko na elementach już pobranych z pamięci DRAM, bez chybień w pamięci podręcznej (dla każdego pobranego wyrazu zachodzić będzie wystarczająca dla danego poziomu pamięci podręcznej lokalność czasowa). Tym sposobem uśredniony czas modyfikacji dla pojedynczej danej (przy odpowiednio dużej liczbie powtórzeń praktycznie niwelujący wpływ pierwszego pobrania z pamięci DRAM), będzie czasem związanym wyłącznie z dostępem do pamięci podręcznej.

Jeśli tablica nie mieści się w pamięci podręcznej, to w pewnym momencie nowe wyrazy tablicy zaczynają podmieniać wyrazy już pobrane. Dla prostego przykładu analizowanej pętli efekt podmian będzie taki, że w miarę zwiększania rozmiaru tablicy liczba chybień w każdym przebiegu pętli będzie rosła, przez co będzie rósł średni czas dostępu do pojedynczej zmiennej. W przypadku istnienia pamięci podręcznej kolejnego poziomu, o większym rozmiarze, tablica wciąż może w całości mieścić się w tej pamięci. W efekcie, czas dostępu do pojedynczej danej stanie się czasem dostępu do pamięci podręcznej kolejnego poziomu. Dla dalej rosnącego rozmiaru tablicy cały proces będzie się powtarzał, aż dla największych tablic czas dostępu stanie się czasem dostępu do pamięci DRAM.

Wyniki wydajnościowe opisanego wyżej eksperymentu obliczeniowego dla stosowanego w książce procesora Intel Core i7-4790 przedstawia rys. 4.8. Na osi poziomej znajduje się rozmiar tablicy wielokrotnie modyfikowanej w pętli, a na osi pionowej czas modyfikacji pojedynczego elementu, obliczony jako iloraz liczby dostępów, wynikającej z kodu źródłowego, podzielonej przez zmierzony czas wielokrotnego wykonania pętli (liczba powtórzeń pętli była w trakcie wykonywania pomiaru inna dla każdego rozmiaru, tak aby uzyskać miarodajne wyniki w rozsądnym czasie). Kolejne rozmiary tablicy na rys. 4.8 są kolejnymi potęgami 2, od 4 kB do 256 MB.

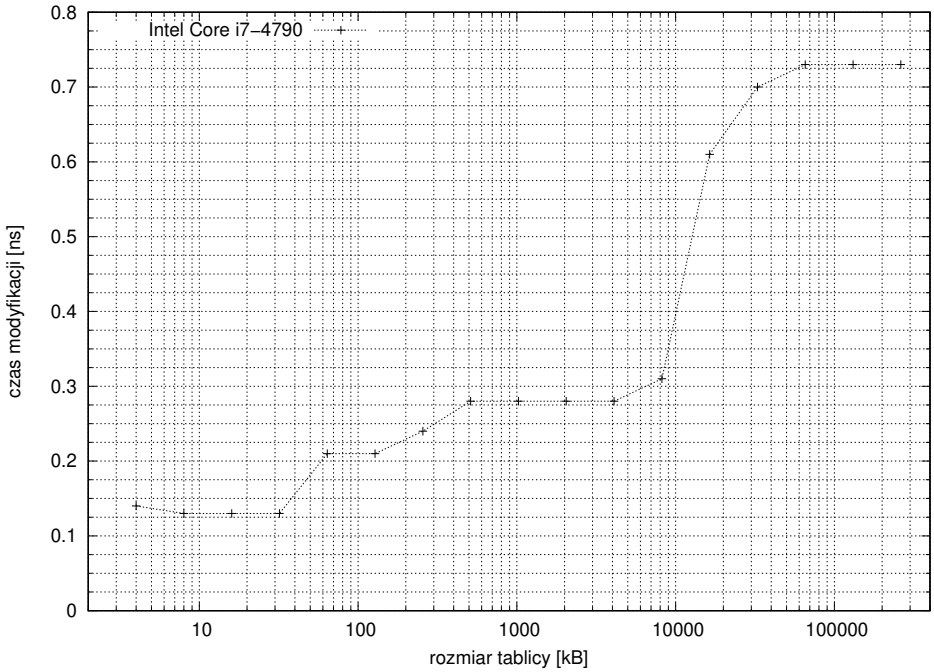
Z rysunku odczytać można dla jakich rozmiarów tablicy następują zmiany pomiędzy stałymi czasami modyfikacji na płaskich fragmentach wykresu, wskazujące na przekroczenie rozmiaru kolejnego poziomu pamięci podręcznej. Widać cztery takie płaskie fragmenty, pierwszy z czasem modyfikacji niewiele ponad 0.1 ns, kończący się w czwartym punkcie wykresu (co przy przyjętych założeniach eksperymentu odpowiada rozmiarowi $2^4 * 2$ kB), drugi, z czasem ok. 0.2 ns i punktem granicznym dla $2^6 * 2$ kB, trzeci z czasem ok. 0.3 ns i końcem dla $2^{12} * 2$ kB, i wreszcie, czwarty, ostatni z czasem ponad 0.7 ns. W efekcie (zakładając standardową praktykę konstruowania pamięci podręcznych z rozmiarami będącymi potęgami 2)[11], dane wykresu prowadzą do następującego oszacowania rozmiarów pamięci: L1 - 32 kB, L2 - 256 kB, L3 - 8 MB. Ostatni płaski fragment wykresu, odpowiadający pamięci DRAM, kończy się dla tablicy o rozmiarze 256 MB – dalszy wzrost rozmiaru może spowodować pewne wahania czasu modyfikacji, związane z innymi mechanizmami funkcjonowania pamięci, takimi jak zarządzanie tablicą stron, rozmiar pamięci TLB itp.

Uzupełniającym badaniem rozmiaru pamięci podręcznych, może być analiza wykorzystująca ten sam eksperyment, ale przeprowadzająca wnioskowanie na podstawie liczby chybień w pamięciach podręcznych, a nie wyników wydajnościowych (choć oczywiście jedno jest powiązane z drugim).

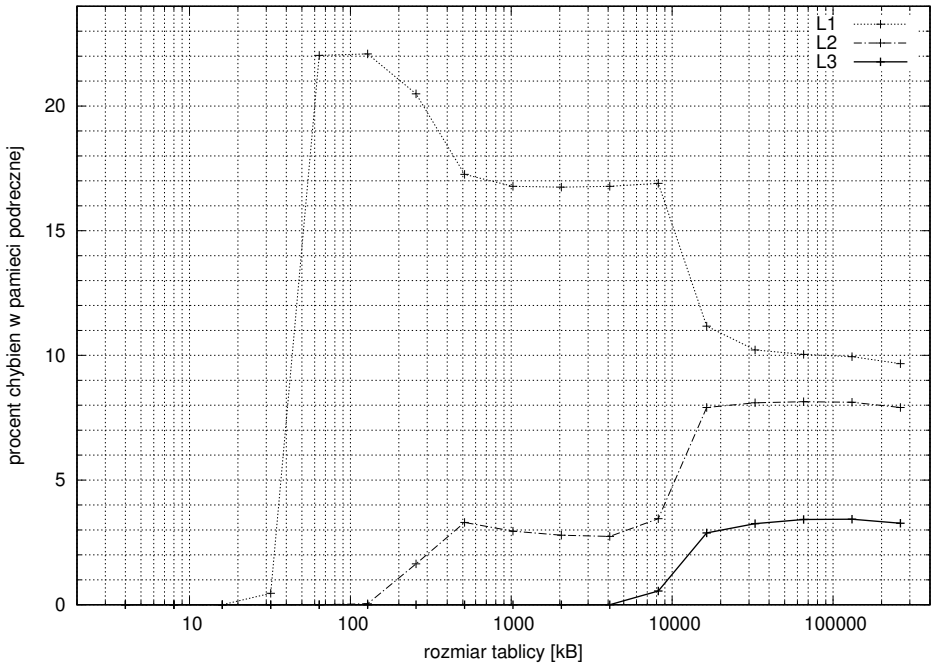
W trakcie wykonania pętli zbierane są dane z liczników sprzętowych dla zdarzeń rozkazów pobrania danych oraz dla chybień w pamięci podręcznej:

- MEM_UOPS_RETIRED.ALL_LOADS,
- MEM_UOPS_RETIRED.L1_MISS,

[11] Praktyka taka dotyczy głównie pamieci L1 i L2, pamięci L3 mogą mieć bardziej zróżnicowane rozmiary



Rysunek 4.8: Średni czas modyfikacji pojedynczego wyrazu tablicy, dla wielokrotnie wykonywanej pętli odwiedzania jej kolejnych wyrazów, w zależności od rozmiaru tablicy.



Rysunek 4.9: Procent chybień w pamięciach podręcznych różnych poziomów, dla wielokrotnie wykonywanej pętli odwiedzania kolejnych wyrazów tablicy, w zależności od rozmiaru tablicy dla procesora Intel Core i7-4790 z rdzeniami o architekturze Haswell

- MEM_UOPS_RETIRED.L2_MISS,
- MEM_UOPS_RETIRED.L3_MISS.

Następnie obliczany jest procent chybień, na podstawie stosunku trzech ostatnich wartości do pierwszej. Rysunek 4.8 przedstawia uzyskane w ten sposób krzywe, jak zwykle dla testowej platformy z mikroprocesorem Intel Core i7-4790.

Wyczerpująca analiza wykresów wymagałaby uwzględnienia szeregu szczegółów funkcjonowania pamięci podręcznych (w tym mechanizmu pamięci zawierających się i odrębnych – *inclusive cache, exclusive cache*, p. 4.2.3 – czy istnienia dodatkowego bufora przechowującego tymczasowo linie pobierane do pamięci podręcznej). Na potrzeby badania rozmiaru pamięci wystarczające jest zaobserwowanie punktów, dla których następuje wzrost procentu chybień dla konkretnego rodzaju pamięci. Widać, że znaczący przyrost, po którym następuje osiągnięcie relatywnie stałej wartości (aż do osiągnięcia rozmiaru kolejnego poziomu pamięci podręcznej) pojawia się przy rozmiarze 32 kB dla pamięci L1, 256 kB dla L2 i 8 MB dla L3, co potwierdza rezultaty osiągnięte w badaniu wydajnościowym.

4.4.2 Eksperymentalne wykrywanie rozmiaru pojedynczej linii pamięci podręcznej

Chcąc znaleźć rozmiar pojedynczej linii pamięci podręcznej można wykonać kolejny eksperyment posługując się ponownie prostym algorytmem:

```
for (j=0; j<rozmiar_tab; j+=skok) tab[j]++;
```

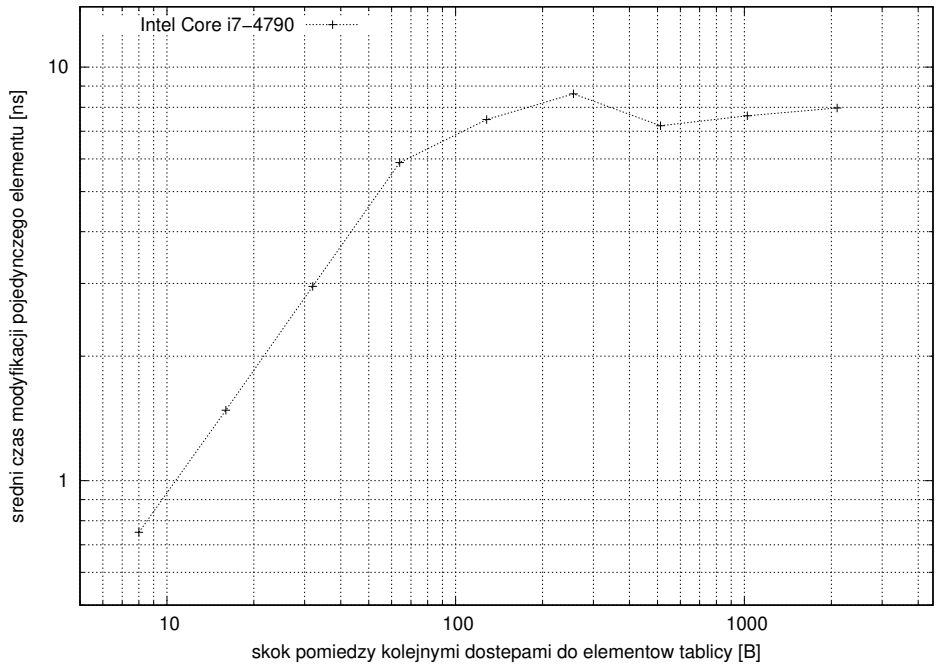
Tym razem, rozmiar tablicy jest stały i na tyle duży, że nie mieści się ona w pamięci podręcznej żadnego z poziomów. Istotny jest rozmiar pojedynczego elementu tablicy, w przeprowadzanych eksperymentach równy 8 bajtom (zmienne podwójnej precyzji). Badanie dotyczy pamięci L1 i polega na przeprowadzeniu serii obliczeń dla rosnących wartości zmiennej `skok`, znowu jako potęg liczby 2.

Średnie czasy modyfikacji pojedynczego elementu tablicy w nanosekundach, dla tablicy o rozmiarze 256 MB, na platformie testowej wyglądają następująco:

`skok`	1	2	4	8	16	32	64	128	256
średni czas modyfikacji	0.75	1.48	2.95	5.88	7.47	8.62	7.22	7.63	7.97

Dla każdej wartości zmiennej `skok` pobranie pierwszego elementu tablicy oznacza chybienie w pamięci podręcznej (wszystkich poziomów) i pobranie linii z pamięci DRAM. W przypadku wartości `skok` równej 1 program wykorzystuje do obliczeń wszystkie pobrane wartości – liczba dostępów wynikająca z kodu źródłowego odpowiada liczbie pobranych elementów. Dla wartości `skok=2` pobrana również zostanie cała linia, ale wykorzystany w programie do modyfikacji będzie tylko co drugi element. Liczba efektywnych dostępów w kodzie będzie dwa razy mniejsza niż liczba pobranych danych. Zakładając ten sam czas pobrania linii co dla `skok=1` oznacza to dwukrotnie wyższy czas przypadający na pojedynczy efektywny dostęp. Podobnie dla wartości `skok=4` – efektywny czas dostępu powinien znów dwukrotnie wzrosnąć. Będzie się tak działo, aż do osiągnięcia przez odstęp pomiędzy dwoma kolejno wykorzystywanymi w pętli elementami (mierzony w bajtach) długości linii pamięci podręcznej (w przypadku obliczeń testowych odstęp jest iloczynem wartość zmiennej `skok` i rozmiaru pojedynczego elementu tablicy).

Kiedy odstęp w pamięci DRAM między kolejno odwiedzanymi elementami przekracza rozmiar linii pamięci podręcznej, efektywnie wykorzystany w programie jest zawsze tylko jeden element tablicy z całej pobranej linii. Tym razem jednak nie wszystkie bloki pamięci DRAM zawierające wyrazy tablicy pobierane są do pamięci podręcznej. Dodatkowo, ze względu na wielodrożność pamięci podręcznej, nie wszystkie jej linie są wykorzystywane. Powoduje to, że czas modyfikacji pojedynczego elementu tablicy



Rysunek 4.10: Średni czas modyfikacji dla pojedynczego elementu tablicy jako funkcja skoku pomiędzy elementami tablicy modyfikowanymi w kolejnych iteracjach pętli

nadal może rosnąć, jednak jego zależność od wartości `skok` przestaje być prostą zależnością liniową, może też w pewnych zakresach być funkcją malejącą.

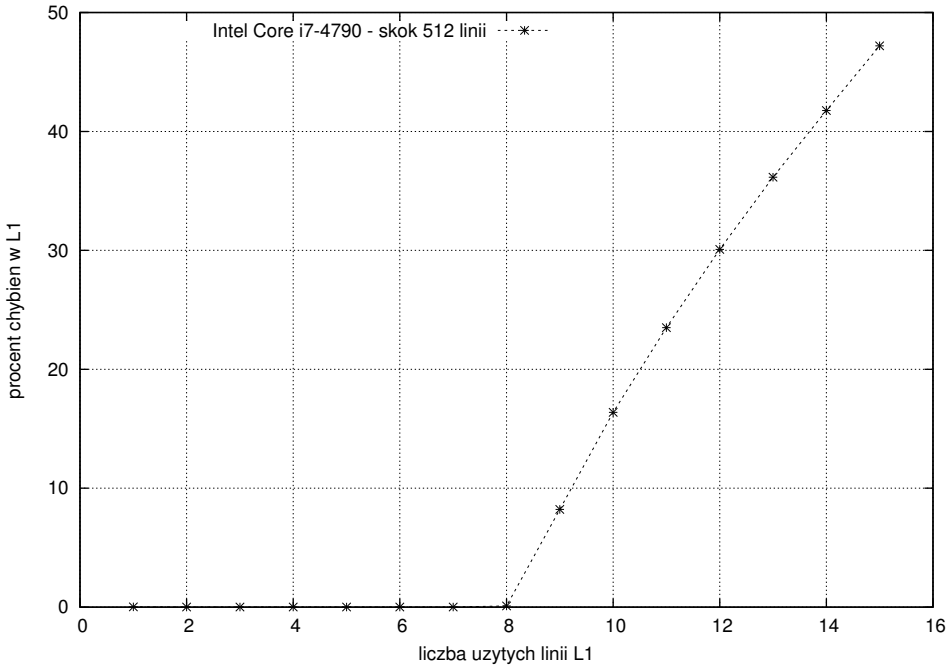
W zamieszczonej powyżej tabeli widać, że wartość graniczna rosnącego liniowo czasu modyfikacji pojedynczej zmiennej (a więc malejącego procentu wykorzystywanych elementów w linii) osiągana jest dla zmiennej `skok` równej 8 (pomiędzy wartością `skok=4` a `skok=8` czas wzrasta nieomal dokładnie dwukrotnie, pomiędzy wartościami 8 i 16 tylko o ok. 30%). Można z tego wyciągnąć wniosek, że długość pojedynczej linii pamięci podręcznej to 8 liczb podwójnej precyzji, czyli 64 bajty.

Charakter powyższych zależności dla platformy testowej dobrze ilustruje wykres na rys. 4.10, odpowiadający danym z zamieszczonej powyżej tabeli. Widać jak początkowo zależność średniego czasu modyfikacji pojedynczego elementu tablicy rośnie liniowo jako funkcja skoku pomiędzy kolejnymi elementami tablicy odwiedzanymi w pętli testowego algorytmu. Na osi x skok jest wyrażony w bajtach co pozwala odczytać rozmiar linii pamięci podręcznej, jako miejsce gdzie krzywa zmienia swój charakter. Po przekroczeniu wartości 64 B następuje spowolnienie wzrostu czasu modyfikacji elementu tablicy, a w dalszej kolejności ustabilizowanie czasu modyfikacji w zakresie ok. 7-9 ns.

Algorytm użyty do badania rozmiaru pamięci podręcznych i rozmiaru pojedynczej linii pamięci podręcznej, mimo swojej prostoty, pozwala na dokładne śledzenie wpływu lokalności odniesień, tak czasowej, jak i przestrzennej, na wydajność dostępów do pamięci.

W pierwszym przypadku lokalność przestrzenna jest zawsze taka sama, niezależnie od rozmiaru tablicy (każda wartość pobrana do pamięci podręcznej jest jednokrotnie wykorzystana w pojedynczym wykonaniu pętli), a wydajność (jako odwrotność czasu dostępu) zależy wyłącznie od lokalności czasowej wynikającej z proporcji rozmiaru tablicy do pojemności pamięci podręcznej konkretnego poziomu.

W drugim przypadku, kiedy z powodu dużego rozmiaru tablicy nie występuje lokalność czasowa, wydajność zależy wyłącznie od lokalności przestrzennej, sterowanej przez wartość zmiennej `skok`.



Rysunek 4.11: Zależność procentu chybień w L1 od liczby używanych bloków pamięci zawierających elementy tablicy przy odwzorowaniu wszystkich bloków w ten sam zbiór linii wielodrożnej pamięci L1

4.4.3 Eksperymentalne wykrywanie drożności pamięci podręcznej

Do zbadania drożności pamięci podręcznej (w badanym przypadku pamięci L1) może posłużyć wersja prostego sumowania wybranych wyrazów tablicy jednowymiarowej, tym razem zapisana jako:

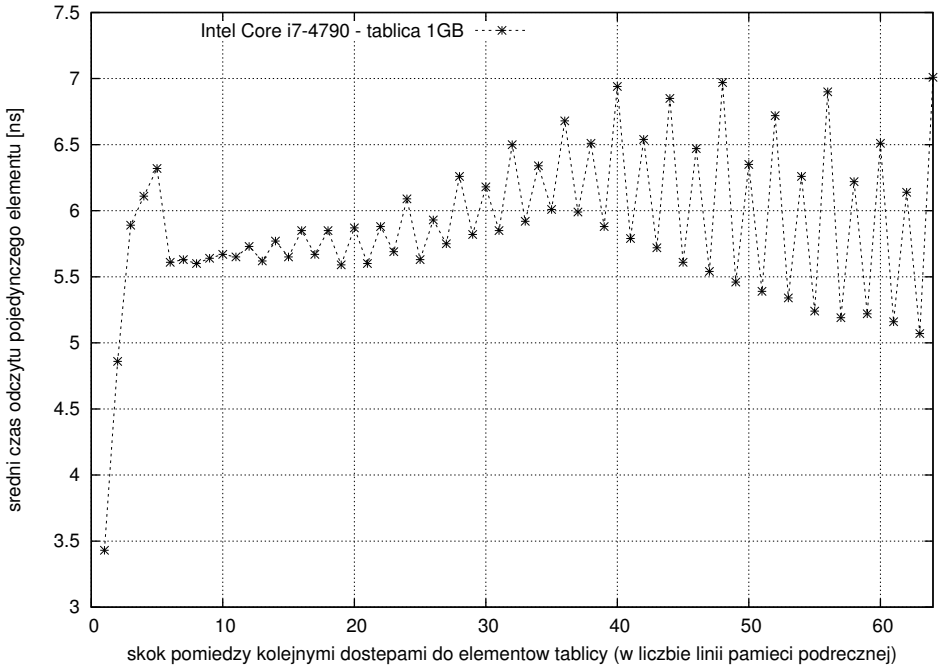
```
for(blok=0; blok < LICZBA_LINII*SKOK; blok+= SKOK){
  suma += a[ blok*(ROZMIAR_LINII)];
}
```

Parametr `ROZMIAR_LINII` odpowiada liczbie elementów tablicy w pojedynczej linii pamięci podręcznej. Dzięki takiej wartości, w każdej iteracji pętli odwiedzany jest inny blok pamięci DRAM, powiązany z pojedynczą linią pamięci. Parametr `SKOK` decyduje o odstępie (liczonym w liczbie bloków, a więc i linii pamięci podręcznej) pomiędzy kolejno odwiedzanymi elementami i zawierającymi je blokami.

W eksperymencie parametry algorytmu dobrane są tak, żeby wszystkie bloki odwzorowane były w ten sam zbiór linii pamięci podręcznej. Dla maszyny testowej wyposażonej w pamięć L1 o pojemności 32 kB (512 linii o rozmiarze 64 B) wystarcza do tego przyjęcie wartości `SKOK = 512`. Przeskok o cały rozmiar pamięci przy odwiedzeniu kolejnego wyrazu tablicy, gwarantuje umieszczenie tego wyrazu (i zawierającego go bloku pamięci DRAM) w tym samym zbiorze linii, co wyrazy poprzedzające.

W celu wykrycia drożności zliczana jest częstość występowania chybień w pamięci L1 (jako proporcja liczby zdarzeń MEM_UOPS_RETIRED.L1_MISS i MEM_UOPS_RETIRED.ALL_LOADS) dla różnych wartości parametru `LICZBA_LINII`. Dla `LICZBA_LINII = 1`, algorytm odczytuje wartości tylko z pierwszego elementu tablicy (do odpowiedniego zbioru linii trafia tylko jeden blok pamięci DRAM). Dla `LICZBA_LINII = 2` odczytywane są dwie wartości z dwóch bloków – obu odwzorowanych w ten sam zbiór linii, dla `LICZBA_LINII = 3` z trzech itd.

Rys. 4.11 przedstawia wyniki przeprowadzonego eksperymentu. Dla pierwszych kilku wartości parametru `LICZBA_LINII` chybienia w pamięci L1 praktycznie nie występują, liczba odwiedzanych



Rysunek 4.12: Zależność czasu dostępu do pojedynczego wyrazu tablicy od skoku pomiędzy dwoma kolejno odwiedzanymi wyrazami – odczyt dla całej tablicy o rozmiarze 1GB

bloków pamięci DRAM jest mniejsza niż liczba linii w pojedynczym zbiorze linii. Nagły wzrost procentu chybień pojawia się po przekroczeniu wartości `LICZBA_LINII = 8`. Prostym wnioskiem jest, że drożność pamięci L1 procesora wynosi 8.

4.4.4 Badanie zależności czasu dostępu do pamięci od wzorca dostępu

Do zilustrowania możliwego wpływu specyficznego wzorca dostępu do pamięci na czas dostępu wykorzystana jest (jak zwykle wykonywana wielokrotnie) ta sama pętla co w punkcie poprzednim:

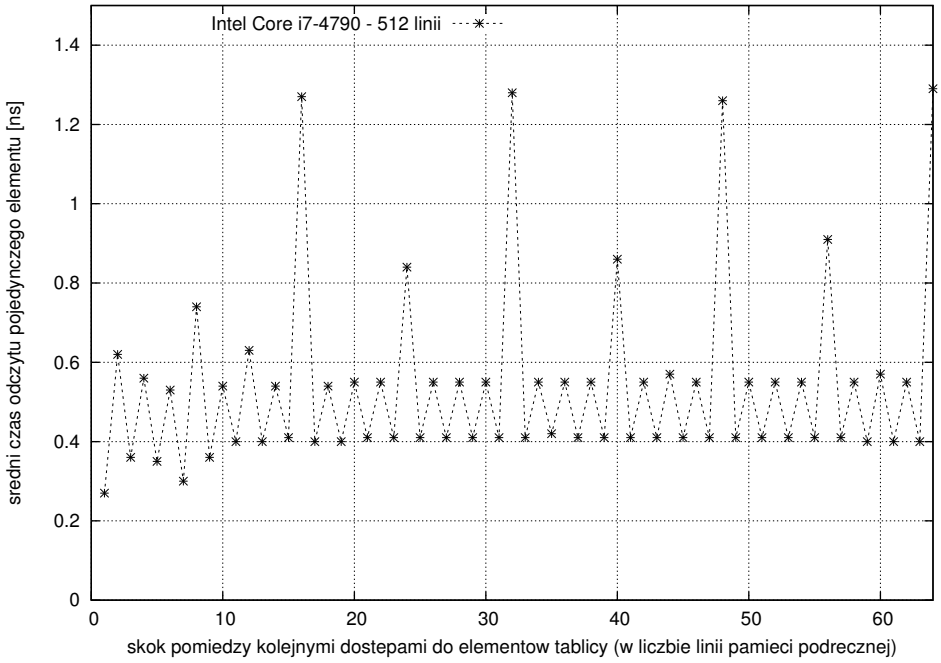
```
for(blok=0; blok < LICZBA_LINII*SKOK; blok+= SKOK){
  suma += a[ blok*(ROZMIAR_LINII)];
}
```

Tablica `a` zawiera liczby podwójnej precyzji, więc parametr `ROZMIAR_LINII` jest, podobnie jak w poprzednim punkcie, równy 8. Parametr `SKOK` jest tym razem podstawową zmienną w eksperymencie, przybierając kolejne wartości od 1 do 64 (1 odpowiada odstępowi 64 B między kolejno odwiedzanymi elementami `a`, 2 skokowi 128 B, itd.).

Pomiary dokonywane są dla odpowiednio długiej tablicy (zaalokowanej z rozmiarem ponad 1 GB), jak zwykle na maszynie z procesorem Intel Core i7-4790 o architekturze Haswell.

Rysunki 4.12 i 4.13 pokazują średni czasu dostępu do pojedynczego wyrazu tablicy (oś *y* na wykresie) dla różnych wartości parametru `SKOK` (oś *x* na wykresie) przy wielokrotnym wykonaniu rozważanej pętli (co daje dodatkowy efekt możliwej lokalności czasowej odniesień).

Wykres 4.12 pokazuje wyniki dla eksperymentu odczytu dla całego zakresu tablicy (liczba odwiedzanych linii, parametr `LICZBA_LINII` w pętli, jest odwrotnie proporcjonalna do skoku pomiędzy dostępami), a wykres 4.13 wyniki dla odczytu z pierwszych 512 linii oddzielonych odpowiednim skokiem (parametr `LICZBA_LINII` w pętli jest równy 512, a więc zakres indeksów odwiedzanych elementów w



Rysunek 4.13: Zależność czasu dostępu do pojedynczego wyrazu tablicy od skoku pomiędzy dwoma kolejno odwiedzanymi wyrazami – wartości w przypadku wielokrotnego odwiedzania pierwszych 512 linii tablicy o rozmiarze 1GB

tablicy rośnie dla kolejnych punktów na osi x, zawsze do 512-krotności skoku odpowiadającego zmiennej `SKOK`).

W pierwszym eksperymencie, przeglądania całego zakresu tablicy, liczba odwiedzanych linii, choć maleje dla kolejnych wartości na osi x, jest na tyle duża, że ich sumaryczny rozmiar każdorazowo kilkakrotnie przekracza rozmiar pamięci L3. Niemniej od pewnej wartości skoku, zjawisko lokalności czasowej dla pamięci L3 zaczyna się pojawiać, co skutkuje malejącym średnim czasem dostępu.

Sumaryczny rozmiar 512 linii w drugim eksperymencie wynosi 32 kB (zakładając długość linii 64 B), co oznacza możliwość zmieszczenia wszystkich odwiedzanych linii w pamięci L1 procesora. Na skutek sposobu funkcjonowania pamięci wielodrożnych i odwzorowania bloków pamięci głównej w różne zbiory linii pamięci podręcznej, zależnie od wartości skoku, nie zawsze wykorzystana jest całą pamięć L1 i pojawiają się chybienia konfliktowe. To samo dotyczy pamięci podręcznych kolejnych poziomów. Nawet dla stosunkowo dużej pamięci L3, specyficzne rozmiary skoków mogą wywołać odwzorowanie w małą liczbę zbiorów linii i chybienia konfliktowe.

Celem eksperymentu nie jest dokładna analiza otrzymanych czasów dostępu do pamięci, wymagająca uwzględnienia szeregu dodatkowych mechanizmów funkcjonowania pamięci. Z punktu widzenia praktyki ważna jest obserwacja czasów dostępu, które dla wybranych przypadków mogą różnić się znacząco, o kilkadziesiąt lub nawet kilkaset procent. W obu eksperymentach, dla odpowiednich zakresów wartości skoku, widać różnicę w czasie dostępu pomiędzy odczytami ze skokiem o parzystą liczbę linii, a odczytami ze skokiem o nieparzystą liczbę linii. Dodatkowo, dla eksperymentu odwiedzania 512 linii szczególnie długim czasem dostępu wyróżniają się dostępy ze skokiem będącym wielokrotnością 8 linii pamięci podręcznej, a zwłaszcza ze skokiem będącym wielokrotnością 16 linii. Prowadzi to do wniosku, że w programach ukierunkowanych na wysoką wydajność obliczeń należy szczególnie zwracać uwagę na sytuacje kiedy w następujących po sobie iteracjach dochodzi do dostępów do tablic w lokalizacjach odległych o wielokrotności charakterystycznych potęg 2.

4.4.5 Przykład przeciwdziałania wzrostowi opóźnień przy dostępach do pamięci poprzez rozciąganie (rozpychanie) tablic (*array padding*)

Kontynuacją badań z poprzedniego punktu jest przykład prostego algorytmu, mającego znaczenie praktyczne i polegającego na wielokrotnym obliczaniu transpozycji macierzy za pomocą pętli:

```
for(i=0;i<n;i++){
  for(j=0;j<n;j++){
    at[i*n+j] = a[j*n+i]; // AT[i][j] = A[j][i]
  }
}
```

W badanym algorytmie występują dwie macierze: będąca źródłem danych macierz A oraz zapisywana macierz A^T, mająca być transpozycją A. W pętli założone jest przechowywanie macierzy wierszami w tablicach jednowymiarowych `a` i `at` (patrz p. 2.1.1). Dla uproszczenia przyjęto macierze kwadratowe *n*x*n*.

Z natury operacji transpozycji, $A^T_{ij} = A_{ji}$, wynika, że wiersze jednej z macierzy są kolumnami drugiej. Jeśli odwiedzamy w standardowej podwójnej pętli wszystkie elementy obu macierzy, to dla jednej z nich odwiedzanie będzie odbywać się wierszami (w kolejnych iteracjach dostęp do kolejnych wyrazów wiersza), a dla drugiej kolumnami (w kolejnych iteracjach dostęp do kolejnych wyrazów kolumny). Praktyczne znaczenie badanego algorytmu polega między innymi na tym, że wnioski z jego analizy dotyczą wszystkich algorytmów, w których z różnych przyczyn wymagany jest dostęp w kolejnych iteracjach pętli do kolejnych elementów kolumny macierzy (mający istotne znaczenie wydajnościowe dla macierzy przechowywanych wierszami, w przypadku macierzy przechowywanych kolumnami będzie to dotyczyć dostępów do kolejnych wyrazów pojedynczego wiersza).

W przykładowej, zaprezentowanej powyżej, implementacji, dostęp wierszami dotyczy macierzy A^T. Dostęp do A polega na tym, że w kolejnych iteracjach wewnętrznej pętli po zmiennej `j`, odczytywane są wyrazy tablicy `a` odległe o *n*, a więc wyrazy w kolejnych wierszach aktualnej *i*-tej kolumny (w rozważanym przypadku `a[j*n+i]` oznacza wyraz A_{ji}).

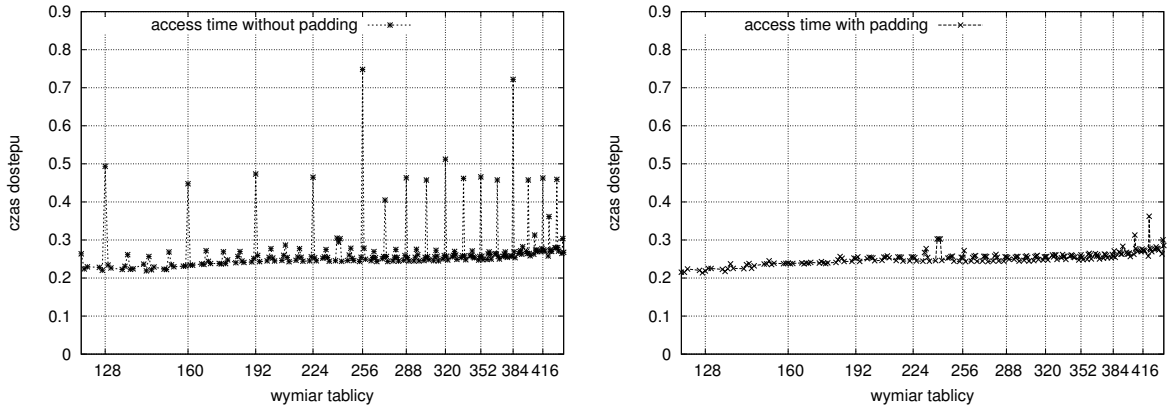
Zgodnie z omawianymi dotychczas zasadami działania pamięci, wielokrotne odczytywanie tablicy ze skokiem pomiędzy kolejno odwiedzonymi wyrazami może stać się źródłem problemów wydajnościowych dla specyficznych wartości skoku. Rys. 4.14 przedstawia po lewej stronie wykres średniego czasu dostępu do pojedynczego wyrazu tablic dla badanego kodu i różnych wymiarów macierzy. Widać, że dla niektórych wymiarów *n* czas ten rośnie znacząco – dwu, a nawet trzykrotnie. Wszystkie te przypadki odpowiadają wymiarom macierzy będącym wielokrotnościami 8 (czyli wierszom o długości będącej wielokrotnością 64 bajtów, a więc szerokości linii pamięci podręcznej).

W celu uniknięcia znaczącego wydłużenia czasu działania dla wybranych rozmiarów tablic można zastosować technikę "rozpychania" tablic omówioną w p. 2.1.1.

Wykorzystywane w tej technice alokowanie zamiast oryginalnej tablicy, tablicy o wydłużonym wierszu powoduje, dla macierzy przechowywanych wierszami, zmianę odstępu pomiędzy kolejnymi wyrazami w kolumnie, a więc zmianę wzorca dostępu do pamięci, w przypadku odwiedzania elementów w tej samej kolumnie.

Zakładając alokację, zamiast tablicy o rozmiarze $n \times n$, tablicy o rozmiarze $n \times (n + o)$, można optymalnie dobrać parametr `o`, tak aby uniknąć niekorzystnego wzorca dostępów do pamięci.

W badanym algorytmie przyjęto, że zamiast dla tablicy $n \times n$, w przypadku kiedy n jest podzielne przez 8, alokuje się pamięć dla tablicy $n \times (n + 1)$ (każdorazowa długość wiersza przechowywana jest w algorytmie w zmiennej `WYMIAR`). W celu realizacji transpozycji, pętle algorytmu pozostają bez zmian, nadal operuje on na *n*n* wyrazach tablic (dodatkowe elementy tablicy `a` mogą nawet pozostać niezainicjowane), zmienia się tylko zapis dostępu do elementów `a`:



Rysunek 4.14: Średni czas dostępu do pojedynczego elementu tablic liczb podwójnej precyzji, podczas wielokrotnego obliczania transpozycji macierzy, dla różnych wymiarów macierzy

```
for(i=0;i<n;i++){
  for(j=0;j<n;j++){
    at[i*n+j]=a[j*WYMIAR+i]; // AT[i][j] = A[j][i]
  }
}
```

Po prawej stronie na rys. 4.14 znajduje się wykres średniego czasu dostępu do pojedynczego wyrazu tablic dla zmodyfikowanego kodu i różnych wymiarów macierzy. Różnice w czasie dostępu dla różnych przypadków są znacząco zmniejszone w stosunku do oryginalnego programu.

Dane zaprezentowane na wykresach dotyczą wielokrotnego dokonywania transpozycji relatywnie małych macierzy, a więc specyficznego przypadku algorytmu. W przypadku jednokrotnego wykonania transpozycji znika znaczące (rzędu kilku razy) zróżnicowanie pomiędzy wydajnością dla poszczególnych zbliżonych wymiarów macierzy. Czasy dostępu nadal zależą od wymiaru, jednak nieznacznie tylko rosną, w momencie kiedy rozmiary tablic przekraczają wartości prowadzące do chybień pojemnościowych dla kolejnych poziomów pamięci podręcznych.

4.5 Pomiary opóźnienia i przepustowości elementów hierarchii pamięci

Celem niniejszego punktu jest poszukiwanie programów, które będą osiągały ekstremalne parametry wydajnościowe – dla danej liczby operacji na pamięci uzyskiwały minimalne i maksymalne czasy wykonania. Z wynikającej z dotychczasowych opisów specyfiki wydajności wykonania programów, gdzie często czynnikiem decydującym o wydajności jest stopień współbieżności działania, badania takie można określić jako poszukiwanie programów, które związane będą z opóźnieniem (*latency*) i maksymalną przepustowością (*throughput*) wykonywania konkretnych operacji na pamięci. Podobne eksperymenty były już opisywane w p. 3.10 dla potoków przetwarzania operacji zmiennoprzecinkowych. Rola takich eksperymentów przy optymalizacji kodu jest intuicyjnie oczywista – pokazują wzorce, których należy unikać i takie, do których należy dążyć.

Jako opóźnienie (przy odczycie z pamięci) można określać czas potrzebny na jednorazowy odczyt danych o najmniejszym możliwym rozmiarze (wyrażany w nanosekundach lub taktach zegara). Przy przeprowadzaniu eksperymentów obliczeniowych dostęp taki, np. do pojedynczej izolowanej zmiennej, trwa zbyt krótko, aby zmierzyć jego czas. Należy więc zaprojektować program, w którym pobierane

będzie wiele danych, ale w taki sposób, żeby utrudnić lub uniemożliwić zastosowanie rozmaitych sprzętowych technik ukrywania opóźnienia (najczęściej związanych z współbieżnością funkcjonowania rozmaitych mechanizmów pamięci).

W praktyce trudno jest zdefiniować opóźnienie przy dostępie do pamięci, w szczególności dla pamięci DRAM, ze względu na to, że w każdym takim dostępie bierze udział szereg elementów sprzętowych, każde o własnych charakterystykach opóźnienia i przepustowości. Z punktu widzenia wykonania kodu zapisanego w języku programowania, dodatkowo poza rozmaitymi własnościami elementów sprzętowych, w uzyskaniu ostatecznej wydajności udział mają np. optymalizacje kompilatora czy zarządzanie pamięcią przez system operacyjny. Próbując określić opóźnienie można rozważać szereg przypadków, w których pojawiają się kolejne mechanizmy wydłużające dostęp, takie jak chybienia w pamięciach podręcznych różnych poziomów, chybienie w pamięci podręcznej tablicy stron (TLB), błąd strony i inne związane z bardziej zaawansowanymi szczegółami sprzętowymi i systemowymi. Duża liczba możliwych mechanizmów i parametrów utrudnia wybór konkretnej ich kombinacji, typowej i mającej znaczenie praktyczne (maksymalny czas odczytów można uzyskać wprowadzając częste większe błędy stron – *major page fault* – ale taka sytuacja rzadko ma miejsce w praktyce).

Prostsza jest sytuacja przy pomiarze przepustowości. Mikrobenchmark powinien pozwalać na wykorzystanie wszystkich technik ukrywania opóźnienia, tak aby sprzęt pracował z maksymalną efektywną wydajnością. Zazwyczaj mechanizmy sprzętowe i systemowe związane z opóźnieniem pojawią się w początkowych chwilach transferu, jednak w miarę wzrostu rozmiaru przesyłanych danych ich efekt będzie malał. Ostateczna przepustowość, wyrażana w bajtach na takt lub na jednostkę czasu (w praktyce najczęściej w Gigabajtach na sekundę, GB/s), będzie charakteryzowała transfery danych o rozmiarach na tyle dużych, że czas opóźnienia będzie zaniedbywalnie mały w stosunku do całkowitego czasu transferu.

Opóźnienie i przepustowość (określana także jako szerokość pasma, *bandwidth*) mogą być określane także na podstawie charakterystyk sprzętu, np. technologii pamięci DRAM i SRAM, częstotliwości taktowania, szerokości (wyrażanej w bitach) magistral łączących poziomy pamięci oraz szczegółowych rozwiązań architektonicznych. Często oszacowania takie dotyczą tylko pewnego podukładu systemu pamięci, a ostateczny efekt wydajnościowy w programie, wynikający ze współpracy wszystkich podukładów, bywa trudny do ustalenia. Z tego względu, wartości teoretyczne są wykorzystywane w dalszych rozdziałach tylko jako wartości referencyjne, wzorcowe, służące do weryfikacji niektórych danych eksperymentalnych, niekoniecznie jako wielkości, które powinny być uzyskiwane w praktyce.

4.5.1 Pomiar opóźnienia przy odczycie danych

Jako punkt wyjścia przy poszukiwaniu kodu mierzącego opóźnienie przy odczycie z pamięci (podręcznej i DRAM) przyjmowana jest po raz kolejny wersja prostej pętli stosowanej w poprzednich punktach, przeprowadzająca sumowanie wartości elementów tablicy (w przypadku rozważanego kodu, z przyczyn omówionych później, przechowującej zmienne całkowite):

```
long int suma = 0;
int indeks = 0;
do {
  suma += tab[indeks];
  indeks++;
} while (indeks <= rozmiar_tablicy);
```

Kod powyższy, równoważny prostej pętli `for`:

```
long int suma = 0;
for (int indeks=0; indeks<rozmiar_tablicy; indeks++) {
```

```
    suma += tab[indeks];
  }
```

umożliwia zastosowanie wielu omawianych dotychczas technik ukrywania opóźnienia. Kompilator może dokonać rozwinięcia pętli (omawianego w p.5.1), może zastosować rozkazy wektorowe, może odwołania do zmiennych skalarnych (o wartościach domyślnie przechowywanych w pamięci) zamienić na odwołania wyłącznie do zawartości rejestrów. Sprzętowo kod może w pełni wykorzystywać możliwości przetwarzania potokowego (w tym dla rozkazów wektorowych), a także szereg innych technik, obejmujących m. in. odpowiednie algorytmy pobierania z wyprzedzeniem i podmiany linii w pamięci podręcznej.

Z punktu widzenia analizy wykonania widać, że w kodzie nie ma lokalności czasowej (każdy element jest wykorzystywany w sumowaniu tylko raz), jednak jest pełna lokalność przestrzenna – dla każdego z elementów tablicy, elementy sąsiadujące z nim są wykorzystywane w obliczeniach. Chcąc zmierzyć czas pojedynczego izolowanego dostępu do pamięci, konieczne jest uwzględnienie czasu pobrania całej linii pamięci podręcznej jako narzutu związanego z pojedynczym dostępem, a więc całkowite usunięcie lokalności przestrzennej.

Celowi temu służy wprowadzenie skoku przy dostępach do pamięci, tak aby każdy dostęp dotyczył odrębnej linii pamięci podręcznej. Modyfikacji ulega jedna linia kodu, w której zwiększana jest wartość indeksu sumowanego elementu:

```
    indeks += skok;
```

gdzie wartości parametru skok przyjmowane są jako równe 16 lub 32, co odpowiada (dla standardowych zmiennych całkowitych) odstępom między kolejnymi sumowanymi elementami równym 64 lub 128 bajtów (co odpowiada typowym rozmiarom linii pamięci podręcznej).

Kod taki traci intuicyjnie oczywistą stosowalność, jednak nadal może zdarzać się w praktyce. Może odpowiadać algorytmowi, który wymaga sumowania wartości tylko wybranych elementów tablicy, może także dotyczyć wartości zapisanych w konkretnym polu zmiennej typu strukturalnego, w sytuacji kiedy sumowanie dotyczy tablicy takich struktur. Pola przechowujące wartości w kolejnych odwiedzanych w pętli strukturach, będą w pamięci oddalone o stałą, zależną od rozmiaru struktury liczbę bajtów.

Likwidacja lokalności przestrzennej przez wprowadzenie skoku w pamięci, wciąż pozostawia możliwość użycia szeregu innych sposobów ukrywania opóźnienia. Kolejną, niezwykle istotną modyfikacją kodu, mającą na celu uniemożliwienie wykorzystania przetwarzania potokowego rozkazów odczytu, jest zastosowanie techniki "ścigania wskaźnika" (*pointer chasing*). W technice tej, indeks elementu tablicy do pobrania w następnej iteracji jest odczytywany z tablicy w iteracji bieżącej. Na poziomie rozważanego kodu źródłowego prowadzi to do pętli:

```
long int suma = 0;
int indeks = 0;
do {
  indeks = tab[indeks];
  suma += indeks;
} while (indeks != 0);
```

Tablica `tab` wymaga teraz odpowiedniego przygotowania do poprawnego działania. W kolejno odczytywanych elementach musi znajdować się ciąg nie powtarzających się indeksów, kończący się zerem. Zakładając, że kolejno odwiedzane elementy mają indeksy zwiększające się o wartość parametru `skok`, prowadzi to do ciągu:

```
0, skok, 2*skok, 3*skok, 4*skok, ..., rozmiar_tablicy, 0
```

(parametr `rozmiar_tablicy` musi być wielokrotnością wartości zmiennej `skok`).

Badana pętla wprowadza nieusuwalną zależność do przetwarzania potokowego rozkazów odczytu z pamięci, co można zweryfikować analizując otrzymany po kompilacji (*gcc*) kod asemblera odpowiadający pętli:

```
.L22:
   movslq (%r15,%rax,4), %rax
   addl %eax, %edx
   testl %eax, %eax
   jne .L22
```

Zawartość rejestru `%rax` odczytywana jest w każdorazowej iteracji pętli z lokalizacji w pamięci o adresie obliczonym na podstawie zawartości `%rax` z poprzedniej iteracji. Odczytana wartość jest dodawana do sumy (przechowywanej w rejestrze `%edx`), a następnie rozkaz `test` sprawdza czy wartość odczytanego indeksu jest zerem, po napotkaniu którego pętla jest przerywana.

Rozważany kod wydaje się daleki od przypadków spotykanych w praktyce, jednak i dla takiej postaci można znaleźć praktyczny odpowiednik. Wystarczy założyć, że mamy do czynienia nie z tablicą struktur, ale z listą struktur, gdzie lokalizacja kolejnego węzła listy jest odczytywana z jednego z pól struktury (sumowaniu podlegać mogłaby wartość innego z pól struktury).

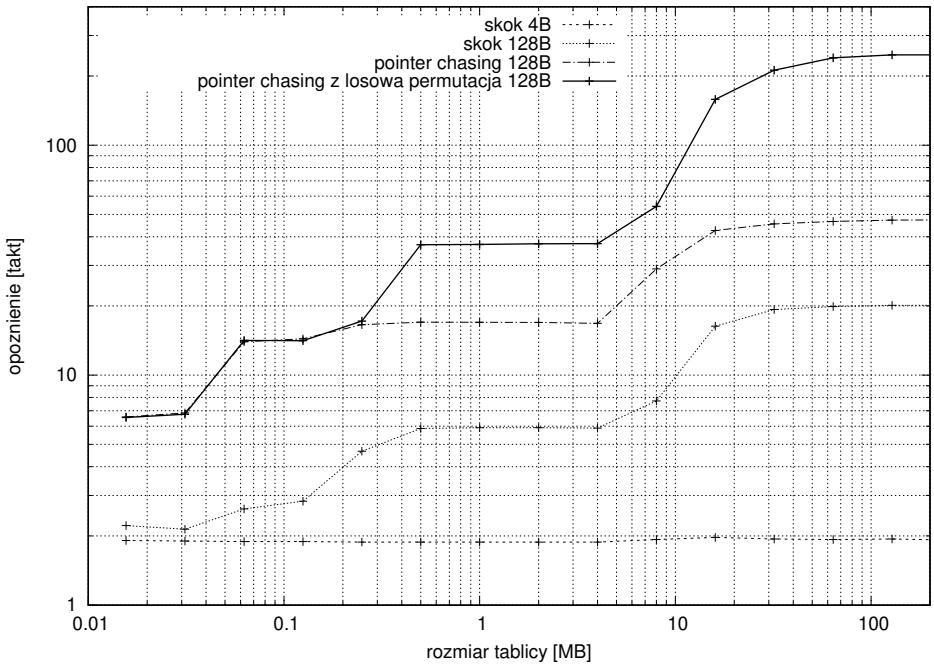
Zastosowanie techniki *pointer chasing* w sposób przedstawiony powyżej nie eliminuje innego ważnego mechanizmu ukrywania opóźnienia: pobierania z wyprzedzeniem. Rozpatrywany ciąg indeksów, nawet dla dużych wartości `skok`, tworzy regularny wzorzec dostępu do pamięci, pozwalający na przewidywanie kolejnych odwiedzanych adresów (dopiero dla bardzo dużych wartości zmiennej `skok`, zazwyczaj odpowiadających odstępom większym od rozmiaru strony pamięci wirtualnej, mechanizm pobierania z wyprzedzeniem przestaje być aktywny).

W celu uniemożliwienia efektywnego działania *prefetchingu* dla dowolnych wartości zmiennej `skok`, zawartość tablicy `tab` jest dodatkowo modyfikowana. Indeksy odwiedzanych elementów zawarte w tablicy są losowo permutowane, dzięki czemu zbiór odwiedzanych adresów pozostaje taki sam, jednak kolejność odwiedzania staje się losowa.

Tak utworzony kod wciąż może odpowiadać stosowanym w praktyce algorytmom. Losowe skoki pomiędzy kolejno odwiedzanymi elementami będą naturalne przy przeglądaniu listy struktur, w przypadku gdy lista tworzona jest przez dodawanie kolejnych węzłów alokowanych niezależnie w różnych miejscach kodu i różnych chwilach jego wykonania. W takiej sytuacji prawdopodobieństwo chaotycznego działania, szczególnie w przypadku braku kontroli nad położeniem w pamięci indywidualnych struktur, wydaje się być wyższe niż sytuacja, kiedy kolejno odwiedzane pola struktur oddalone są o stałą liczbę bajtów.

Dla opisanych powyżej wariantów pętli można przeprowadzać pomiary opóźnienia dla tablic o różnych rozmiarach, dzięki czemu uzyskane zostaną wyniki dla różnych poziomów w hierarchii pamięci. Wykres na rys. 4.15 przedstawia (dla procesora Intel Core i7-4790 z rdzeniami o architekturze Haswell) wyniki takich pomiarów dla rozmiarów tablicy `tab` od najmniejszej wartości odpowiadającej 16 kB (tablica w całości mieszcząca się w pamięci podręcznej L1) do wartości maksymalnej odpowiadającej 128 MB, większej od rozmiaru pamięci L3 (w kodzie tablica `tab` jest alokowana jednorazowo jako odpowiednio duża dla wszystkich badanych zakresów indeksów).

Rozmiar tablicy w MB znajduje się na osi x wykresu, natomiast na osi y zaznaczone jest opóźnienie wyrażone w liczbie taktów na pojedynczy odczyt liczby całkowitej (obliczone na podstawie czasu odczytu w nanosekundach i częstotliwości pracy rdzenia w GHz). Cztery krzywe na wykresie odpowiadają czterem wariantom pętli: standardowemu obliczaniu sumy kolejnych elementów tablicy (”skok 4B”), sumowaniu wartości wyrazów odległych o `skok` elementów (”skok 128B”) oraz dwóm przypadkom sumowania w przypadku mechanizmu *pointer chasing* - bez permutacji ("pointer chasing 128B") i z



Rysunek 4.15: Uzyskany eksperymentalnie czas opóźnienia przy odczycie z pamięci różnych poziomów dla pojedynczego rdzenia o mikroarchitekturze Intel Haswell

permutacją indeksów ("pointer chasing z losowa permutacja 128B"). Jak wskazują ich nazwy, dla każdej z trzech ostatnich krzywych zastosowano najmniejszy odstęp pomiędzy odwiedzanymi elementami (determinowany przez wartość zmiennej `skok`) równy 128 bajtów.

Z wykresu odczytać można kilka ciekawych faktów dotyczących funkcjonowania pamięci. Dla standardowej pętli obliczania sumy kolejnych wyrazów tablicy, lokalność czasowa i rozmaite techniki ukrywania opóźnienia powodują, że czas dostępu do pojedynczego elementu tablicy jest praktycznie stały, równy w przybliżeniu dwóm taktom, niezależnie od rozmiaru tablicy. Oznacza to, że niezależnie od tego czy tablica mieści się w pamięci podręcznej dowolnego poziomu, czy w pamięci DRAM, opóźnienie jest takie samo, co świadczy o zrównoważonym doborze technik ukrywania opóźnienia dla różnych poziomów hierarchii pamięci. Opóźnienie dwóch taktów, dla rozważanego procesora, jest równoważne czasowi dostępu ok. 0,5 ns i przepustowości lekko przekraczającej 8 GB/s. W przypadku pamięci DRAM są to wartości wysokie, gwarantujące sprawną realizację pętli.

Brak lokalności przestrzennej (dla skoku 128 bajtów) wprowadza już rozróżnienie czasów dostępu dla różnych rozmiarów tablicy. Dostępy do tablic zawartych w całości w pamięci podręcznej L1 nadal mają opóźnienie zbliżone do dwóch taktów, jednak dla większych tablic zaczynają odgrywać rolę częste podmiany linii pomiędzy poziomami pamięci podręcznej, a dla największych tablic także pamięcią DRAM, co prowadzi do rosnących czasów dostępu. Opóźnienie zwiększa się do około 3 taktów dla pamięci L2, 6 taktów dla L3 i aż do ok. 20 taktów dla DRAM. Widać, że wzrosty są różne dla różnych poziomów pamięci, im dalej od rdzeni procesora tym większe (dla pamięci DRAM czas dostępu rośnie ponad dziesięciokrotnie). Wciąż jednak są to wzrosty relatywnie małe, odpowiadające np. czasom dostępu dla pamięci DRAM ok. 5 ns, co jest równoważne przepustowości ok. 0,8 GB/s.

Znaczący przyrost czasów dostępu przynosi technika *pointer chasing*. Praktyczne uniemożliwienie przetwarzania potokowego prowadzi do opóźnień ponad 6 taktów dla L1 i ok. 14 taktów dla L2. Dla L2 zwiększenie opóźnienia jest ponad pięciokrotne, dla pozostałych poziomów hierarchii pamięci ok. trzykrotne (dla L3 czasy dostępu zbliżają się do 17 taktów, dla DRAM przekraczają 45 taktów).

Wprowadzenie losowej permutacji indeksów i chaotycznego dostępu do tablic nie zwiększa czasów dostępu dla pamięci podręcznej dwóch pierwszych poziomów, L1 i L2. Można wywnioskować, że już dla standardowej techniki *pointer chasing* osiągają one wartości maksymalne dla istniejących mechanizmów sprzętowych. Inaczej dzieje się w przypadku pamięci L3 i DRAM – uniemożliwienie efektywnego pobierania z wyprzedzeniem prowadzi do dalszego wzrostu opóźnienia. Dla L3 do wartości ok. 37 taktów (9 ns), dla DRAM aż do ponad 240 taktów (60 ns). Dla pamięci DRAM oznacza to ponad 100-krotny wzrost czasu dostępu w porównaniu ze standardową pętlą sumowania wyrazów tablicy. Dla pamięci podręcznych przybliżone wzrosty są mniejsze: 3,5-krotny dla L1, 7-krotny dla L2 i 20-krotny dla L3.

W tym momencie porównać można uzyskane eksperymentalnie najwyższe wartości opóźnienia z danymi teoretycznymi, podawanymi jako charakterystyki sprzętu. Dokumentacja producenta mikroprocesora Intel Core i7-4790 dla mikroarchitektury Haswell podaje następujące wartości opóźnienia dla kolejnych poziomów pamięci podręcznej: L1 – 4-6 taktów, L2 – 11-12 taktów, L3 - co najmniej 34 takty (choć dla zbliżonej, różniącej się głównie procesem wytwarzania, architektury Broadwell dokumentacja podaje 50-60 taktów). Wartości teoretyczne są jak widać zbliżone do uzyskanych eksperymentalnie, choć należy zaznaczyć, że dla pamięci L3, będącej pamięcią wspólną dla rdzeni, opóźnienie zależeć może od architektury całego mikroprocesora wielordzeniowego.

Na opóźnienie wpływ może mieć także funkcjonowanie innych elementów rdzenia i mikroprocesora, np. pamięci podręcznej tablicy stron, TLB. Narzut związany z pamięcią TLB będzie rósł w przypadku większych tablic i większych minimalnych odstępów między odwiedzanymi elementami tablicy (w eksperymentach np. dla tablic większych od rozmiaru L2 i mieszczących się w pamięci L3 oraz dla wartości zmiennej `skok` odpowiadającej odstępom rzędu rozmiaru strony pamięci wirtualnej, co zwiększa liczbę odwiedzanych stron i częstotliwość podmian linii w pamięci TLB, opóźnienie dla pamięci L3 wzrastało powyżej 50 taktów, czyli o ponad 30%).

W przypadku pamięci DRAM wpływ na opóźnienie ma sposób działania szeregu elementów poza rdzeniem, np. magistrali łączącej pamięć z procesorem oraz samych kości pamięci. Uzyskane wartości opóźnienia, odpowiadające czasom dostępu rzędu kilkudziesięciu nanosekund (w konkretnym przypadku przeprowadzanego eksperymentu 60 ns), są typowe dla współczesnych układów procesor-pamięć DRAM.

Przeprowadzony test wskazuje, że mimo istnienia wielu złożonych mechanizmów dostępu do hierarchii pamięci, udaje się wyróżnić wśród nich kilka podstawowych, decydujących o ukrywaniu opóźnienia, takich jak potokowe przetwarzanie rozkazów dostępu, lokalność czasowa i przestrzenna odniesień, *prefetching*. Odpowiednio napisany program może w zdecydowanym stopniu spowodować neutralizację tych mechanizmów w trakcie wykonania.

4.5.2 Prawo Little'a dla dostępów do pamięci

Omawiane w p.3.8 prawo Little'a wiąże opóźnienie i przepustowość przetwarzania przez pewien układ, którego praca charakteryzuje się określonym stopniem współbieżności, czyli liczbą współbieżnie realizowanych operacji (rozkazów, instrukcji). W przypadku przetwarzania przez układ z maksymalną wydajnością, zgodnie z prawem Little'a, stopień współbieżności jest iloczynem opóźnienia i przepustowości.

W badaniu i optymalizacji wydajności pobierania z pamięci, podobnie jak w przypadku przetwarzania rozkazów przez potoki, parametr opóźnienia jest traktowany jako zadany, wynikający z technologii i organizacji pracy sprzętu. Optymalizacja, jako dążenie do maksymalizacji przepustowości, musi dążyć do zapewnienia wystarczająco dużego stopnia współbieżności.

Jakiego rzędu powinien być taki stopień współbieżności? Proste oszacowanie, bazujące na wynikach dla pamięci DRAM uzyskanych w poprzednim punkcie (czas dostępu ok. 60 ns) oraz wymaganej

przepustowości rzędu np. 20 GB/s prowadzi do wyniku:

$$60[ns] \cdot 20[GB/s] = 1200[B]$$

Oznacza to, że w celu uzyskania maksymalnej przepustowości odczytów z pamięci dla pojedynczego rdzenia, współbieżnie powinny być generowane rozkazy pobrania dla ok. 1200 bajtów. Zakładając, że pojedyncze żądanie dostępu do np. elementu tablicy jest równoważne żądaniu pobrania linii pamięci podręcznej, przy dodatkowym założeniu pełnej lokalności przestrzennej i późniejszym wykorzystaniu wszystkich danych z linii, nadal oznacza to ponad 1200/64=18,75 współbieżnych żądań pobrania różnych linii pamięci podręcznej.

Konstruując mikrobenchmarki z pętlą pobierania danych należy zwrócić uwagę, że użycie standardowej pętli i tylko jednej tablicy z jednorazowym dostępem w iteracji, jak np. w testach w poprzednim punkcie, nawet zakładając efektywne użycie przez procesor pobierania z wyprzedzeniem, może nie wystarczyć do uzyskania wymaganego stopnia współbieżności. Dlatego chcąc maksymalizować przepustowość pobierania danych z pamięci powinno się zapewnić generowanie przez kompilator wielu żądań dostępu w każdej iteracji pętli. Można to uzyskać np. poprzez użycie wielu tablic, z których odczytywane będą dane, ewentualne odczyty z wielu miejsc pojedynczej tablicy w każdej iteracji (dodatkowo umożliwiając kompilatorowi rozwiniecie pętli – patrz p. 5.1).

4.5.3 Pomiar przepustowości przy odczycie danych

Znając kod, uzyskany w p. 4.5.1, uniemożliwiający funkcjonowanie mechanizmów ukrywania opóźnienia, można próbować zaprojektować program, gdzie mechanizmy te będą mogły skutecznie zadziałać, prowadząc do uzyskania przy odczytach z pamięci wydajności zbliżonych do teoretycznych maksimów przepustowości (szerokości pasma, *bandwidth*).

Kodem wykorzystywanym przy testowaniu jest kolejna wersja prostej pętli używanej w poprzednich pomiarach, tym razem z czterema tablicami:

```
for (k=0; k<liczba_powtorzen; k++) {
  for (j=0; j<rozmiar_tablic; j++) {
    suma += a[j]*b[j] + d[j]*c[j];
  }
}
```

Skok pomiędzy odwiedzanymi elementami tablic, mierzony w liczbie elementów tablicy, jest równy 1, w celu jawnego zagwarantowania lokalności przestrzennej. W efekcie każdy pobrany z pamięci bajt (zakładając, że pobieranie z wyprzedzeniem poprawnie rozpozna wzorzec dostępu do pamięci i nie będzie pobierać danych niewykorzystywanych w programie) dotyczy zmiennych używanych w kodzie i efektywna wydajność na potrzeby programu pokrywa się z wydajnością pracy sprzętu.

Parametr `rozmiar_tablic` zmienia się podobnie jak w poprzednich badaniach w taki sposób, aby realizować pomiary dla różnych poziomów pamięci. W celu umożliwienia porównania z testem opóźnienia, tablica zawiera zmienne typu *float*, o rozmiarze 32 bajty, podobnie jak zmienne całkowite.

W ostatecznym efekcie uzyskuje się kod, w którym zmaksymalizowana jest lokalność przestrzenna, a dostępy do pamięci tworzą regularny wzorzec pozwalający na sprawne funkcjonowanie pobierania z wyprzedzeniem. Ostatnim z czynników jest dążenie do wygenerowania jak największej liczby żądań dostępu w jednostce czasu (w efekcie także na pojedynczy takt), zgodnie z analizą przeprowadzoną w poprzednim punkcie na podstawie prawa Little'a. Celowi temu służy wprowadzenie czterech tablic, odpowiednio wyrównanych w pamięci, oraz wykorzystanie możliwości optymalizacyjnych kompilatora, co w praktyce okazuje się wystarczające do osiągnięcia wydajności zbliżonych do teoretycznego maksimum.

Kod asemblera, podobnie jak w przypadku innych testów, jest badany przed realizacją obliczeń, w celu sprawdzenia optymalizacji zastosowanych przez kompilator. Przykładowy kod, utworzony przez kompilator *icc* dla wewnętrznej pętli benchmarku, wygląda następująco:

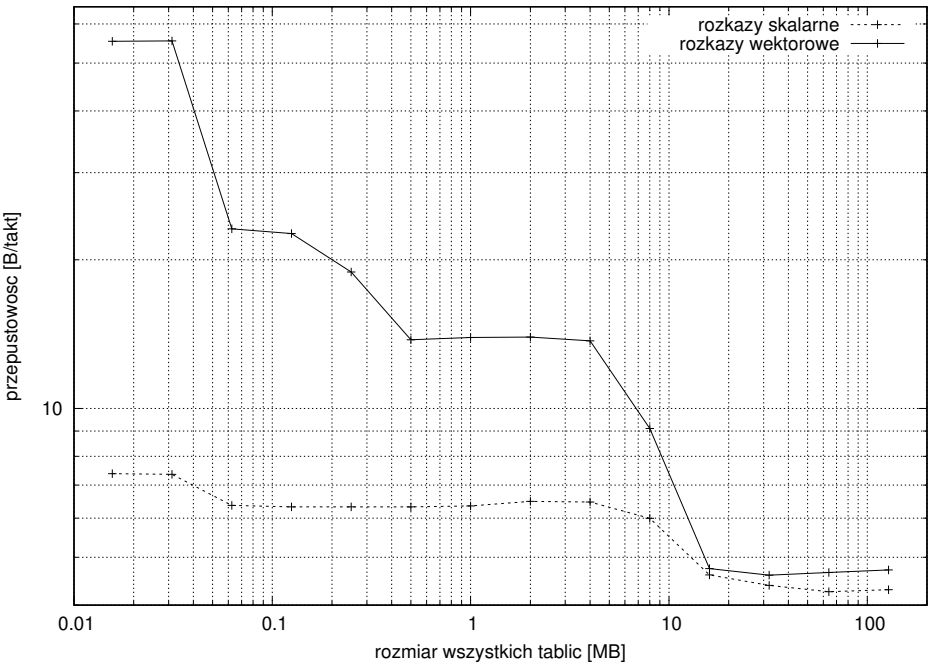
```
..B1.46:
        vmovups   (%rsi,%rcx,4), %ymm3
        vmovups   32(%rsi,%rcx,4), %ymm6
        vmovups   (%rdi,%rcx,4), %ymm4
        vmovups   32(%rdi,%rcx,4), %ymm7
        vmulps    (%rbx,%rcx,4), %ymm3, %ymm5
        vmulps    32(%rbx,%rcx,4), %ymm6, %ymm8
        vfmadd231ps (%r8,%rcx,4), %ymm4, %ymm5
        vfmadd231ps 32(%r8,%rcx,4), %ymm7, %ymm8
        vaddps    %ymm2, %ymm5, %ymm2
        vaddps    %ymm1, %ymm8, %ymm1
        addq      $16, %rcx
        cmpq      %rdx, %rcx
        jb        ..B1.46
```

Pierwsza obserwacja potwierdza poprawne zadziałanie opcji kompilacji *-march=core-avx2*, wymuszającej wektoryzację kodu (co jest szczególnie ważne, ponieważ kompilator musi ostatecznie dokonać redukcji wszystkich sumowanych wartości do pojedynczej zmiennej *suma* – w przypadku przedstawianego kodu jest to realizowane poza pętlą). Użyte są 256-bitowe rejestry *ymm*. Efektywne działanie rozkazów wykonania operacji arytmetycznych (*vmulps*, *vfmadd231ps*, *vaddps*) powinno spowodować ukrycie czasu ich realizacji wewnątrz czasu pobierania danych. Dalsza analiza kodu asemblera pokazuje, że w pojedynczej iteracji pętli pobierane jest 256 bajtów danych (każde odniesienie do pamięci w kodzie oznacza pobranie 32 bajtów) i wykonywane są 64 pojedyncze operacje dodawania i mnożenia na liczbach zmiennoprzecinkowych. W kodzie źródłowym, w pojedynczej iteracji znajdują się tylko 4 operacje w pojedynczej linii kodu, co oznacza, że kompilator dokonał rozwinięcia wewnętrznej pętli o czynnik 8[12], dodatkowo dwukrotnie grupując operacje w kolejnych 4 liniach w rozkazy wektorowe. Dokładnie tego typu działania są wymagane w celu maksymalizacji współbieżności przy wykonaniu kodu, a więc generowania jak największej liczby żądań dostępu do pamięci w jednostce czasu.

Efekt wydajnościowy wykonania tak zaprojektowanego kodu przedstawia wykres na rys. 4.16. Wykres zawiera dwie krzywe – jedna z nich odnosi się do opcji kompilacji z wektoryzacją i wykonania omawianego wyżej kodu asemblera, natomiast druga odpowiada wykonaniu kodu przy użyciu rozkazów skalarnych. Wydajność każdorazowo wyrażana jest w B/takt, co pozwala uniezależnić ją od zmiennej częstotliwości pracy rdzenia.

Na osi x, standardowo, znajduje się rozmiar, tym razem wszystkich tablic, wyrażany w MB, pozwalający na odczyt przepustowości dla różnych poziomów w hierarchii pamięci. Poza charakterystycznymi, występującymi już wcześniej, różnicami w przepustowości dla różnych poziomów pamięci, na wykresie z rys. 4.16 pojawiają się znaczące różnice w przepustowości dla kodu binarnego korzystającego z rozkazów wektorowych i kodu skalarnego. Różnice te są szczególnie duże dla pamięci L1, mniejsze dla L2, jeszcze mniejsze dla L3, osiągając najmniejsze wartości dla pamięci DRAM (dla wielu architektur te ostatnie różnice praktycznie znikają, zrównując wydajność kodu wektorowego i skalarnego). Znajomość tej specyficznej cechy układu pamięci może mieć istotne znaczenie przy optymalizacji kodu korzystającego z hierarchii pamięci, w szczególności przy istotnym udziale wykorzystania pamięci podręcznej bliskiej potokom przetwarzania (poziomy L1 i L2).

[12]Optymalizacja rozwijania pętli (*loop unrolling*) omawiana jest w p.5.1



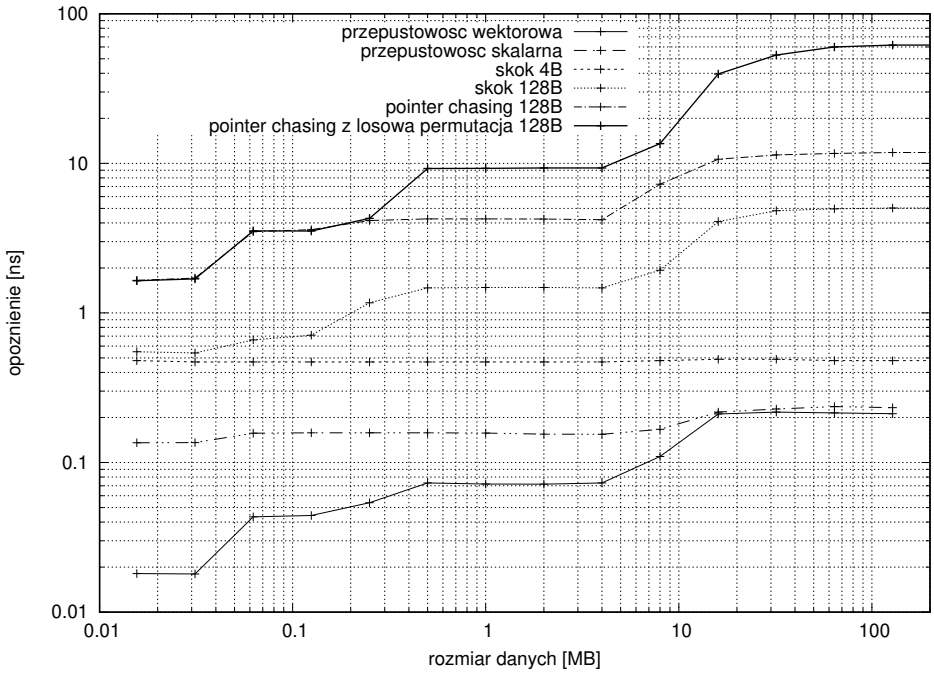
Rysunek 4.16: Uzyskana eksperymentalnie maksymalna przepustowość pamięci różnych poziomów dla pojedynczego rdzenia o mikroarchitekturze Intel Haswell (mikroprocesor Core i7-4790)

Porównanie z wartościami teoretycznymi pokazuje jak daleko od możliwości czysto sprzętowych znajdują się uzyskane eksperymentalnie wyniki. Dla pamięci L1 dane producenta dla odczytu danych wynoszą 64 B/takt, podczas gdy w eksperymencie uzyskane zostało 55 B/takt, co stanowi ok. 86% teoretycznego maksimum. Być może dalsze optymalizacje, np. na poziomie kodu asemblera pozwoliłyby na zwiększenie wydajności, jednak w tym wypadku, jak i w dalszej części książki, uzyskanie wydajności zbliżonej do 90% maksymalnej teoretycznej uznawane jest za wystarczające. W wielu przypadkach przy optymalizacji wydajności programów potwierdza się reguła ”malejących zysków” – osiągnięcie kolejnych drobnych przyrostów wydajności okupione bywa znacznym nakładem związanym z analizą wykonania programu i poszukiwaniem sposobów jego optymalizacji.

Dla pamięci podręcznej L2 (funkcjonującej jako pamięć odrębna, *exclusive cache*) dokumentacja producenta podaje dla rdzeni Haswell wartość maksymalnej teoretycznej przepustowości 64 B/takt. Jednak już dla zbliżonej architektury Broadwell poza pojęciem maksymalnej teoretycznej przepustowości (*peak bandwidth*) wprowadza także pojęcie przepustowości długotrwałej (*sustained bandwidth*), wynoszącej 32 B/takt. Osiągnięte eksperymentalnie wyniki, ok. 23 B/takt, są zdecydowanie bardziej bliskie tej ostatniej wielkości.

Podobnie ma się sprawa z pamięcią L3. Dokumentacja producenta podaje parametry tylko dla mikroarchitektury Broadwell, także w dwóch wariantach: teoretyczne maksimum 16 B/takt i długotrwałą przepustowość 14 B/takt. Osiągnięte w teście wyniki, zbliżone do 14 B/takt, nieomalże dokładnie pokrywają się z tą drugą wartością.

Wyniki dla pamięci DRAM w najmniejszym stopniu są zróżnicowane w zależności od tego czy rozkazy dostępu do pamięci są skalarne czy wektorowe. W obu przypadkach wynoszą ok. 4,5 B/takt, co odpowiada przepustowości ok. 18.5 GB/s. Podobnie jak w przypadku opóźnienia w dostępie do pamięci, wyniki zależą od wielu układów poza rdzeniem. Warto jednak zwrócić uwagę, że osiągnięte w teście kodu jednowątkowego, korzystającego z tylko jednego rdzenia, wartości przepustowości osiągają ponad 70% maksymalnej teoretycznej przepustowości dla całego mikroprocesora (25,6 GB/s).



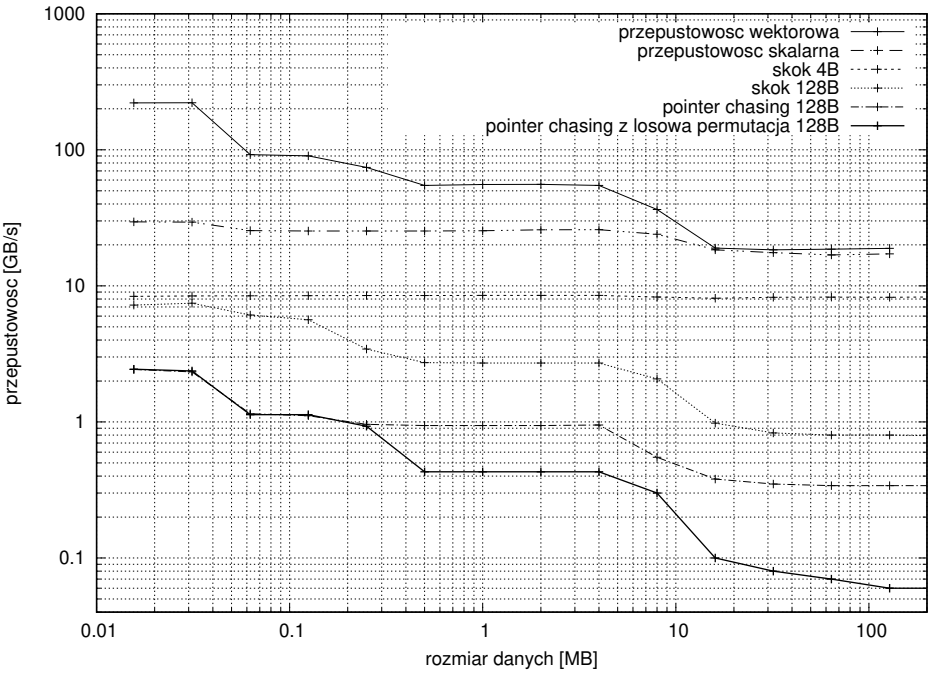
Rysunek 4.17: Zestawienie uzyskanych czasów odczytu pojedynczej zmiennej w przeprowadzonych eksperymentach dla pamięci różnych poziomów pojedynczego rdzenia o mikroarchitekturze Haswell w mikroprocesorze Intel Core i7-4790

4.5.4 Zakres maksymalnej i minimalnej wydajności dostępów do pamięci

Podsumowaniem wyników pomiarów opóźnienia i przepustowości przy dostępach do hierarchii pamięci jest zestawienie najważniejszych osiągniętych rezultatów na wykresach 4.17 i 4.18. Znajdują się na nich krzywe odpowiadające opisanym wcześniej eksperymentom, jednak wyniki podane są w miarach częściej spotykanych w praktyce analizy wydajności programów: na wykresie 4.17 jest to czas dostępu do pojedynczej zmiennej wyrażony w nanosekundach, a na wykresie 4.18 przepustowość wyrażona w GB/s. Celem zestawień i analiz zawartych w niniejszym punkcie jest zwrócenie uwagi na zakres możliwych do uzyskania w praktyce wydajności pamięci różnych poziomów oraz na pewne dodatkowe uwarunkowania dotyczące uzyskiwanych wydajności. Analizy dotyczą uzyskanych danych eksperymentalnych, a więc konkretnego mikroprocesora w konkretnym komputerze, w przypadku innych charakterystyk sprzętowych i systemowych sposób analizy powinien zachować swoją ważność, choć wyniki i wnioski mogą być odmienne.

Na każdym z wykresów znajduje się sześć linii o nazwach odnoszących się do typu eksperymentu, w którym uzyskane zostały przedstawione wyniki wydajnościowe. Linia ”przepustowosc wektorowa” odpowiada eksperymentowi pomiaru maksymalnej przepustowości przy użyciu wektorowych rozkazów odczytu, ”przepustowosc skalarna” do tego samego eksperymentu, ale z użyciem rozkazów skalarnych. Obie linie ”skok 4B” i ”skok 128B” odnoszą się do standardowego (liniowego w sensie niekorzystania z pośredniego adresowania *pointer chasing*) odczytu z tablicy ze skokiem odpowiednio 4 bajty i 128 bajtów. Dwie ostatnie linie odpowiadają eksperymentom z użyciem techniki *pointer chasing*, pierwsza, ”pointer chasing 128B”, eksperymentowi dostępów do kolejnych wyrazów tablicy oddalonych o 128 bajtów, ostatnia, szósta, ”pointer chasing z losowa permutacja 128B”, zgodnie ze swoją nazwą takim samym warunkom jak dla linii piątej z dodatkową losową permutacją odczytywanych elementów tablicy.

Na wykresach widoczne są różnice w charakterystykach wydajności dla pamięci różnych poziomów,



Rysunek 4.18: Zestawienie uzyskanych przepustowości w przeprowadzonych eksperymentach dla pamięci różnych poziomów pojedynczego rdzenia o mikroarchitekturze Haswell w mikroprocesorze Intel Core i7-4790

stąd analiza wykresów zostanie dokonana odrębnie dla każdego z nich. Jako podstawowa przyjęta będzie miara czasu dostępu (pozwalająca łatwo uzyskać czas wykonania programu jako iloczyn czasu dostępu, w konkretnym przypadku przeprowadzonych eksperymentów czas odczytu, oraz liczbę odczytywanych wartości zmiennych). Każdorazowo miara czasu dostępu przeliczana będzie także na wydajność w GB/s, jako miarę często pojawiającą się w badaniach wydajności.

Pamięć podręczna poziomu L1 charakteryzuje się czasami odczytu od ok. 0,018 ns do ok. 1,67 ns, co daje stosunek wydajności ok. 90 (z wydajnościami w GB/s od 2,4 do 221). Poza samym zakresem wydajności pokazującym skalę możliwych zysków i strat, ciekawa jest specyfika różnych typów dostępu. Okazuje się, że w przypadku L1 losowa permutacja dla techniki *pointer chasing* nie odgrywa istotnej roli – pobieranie z wyprzedzeniem, *prefetching*, nie funkcjonuje dla L1, programista nie musi się przejmować układaniem dostępów do L1 w regularne wzorce.

Znacząca okazuje się, podobnie jak dla wszystkich pozostałych poziomów pamięci możliwość przetwarzania potokowego rozkazów odczytu – zastosowanie standardowych dostępów do tablicy, w miejsce *pointer chasing* zwiększa wydajność ponad trzykrotnie. Bez istotnego znaczenia okazuje się skok pomiędzy odwiedzanymi w kolejnych iteracjach pętli elementami pojedynczej tablicy – dla skoku 4 bajty czas odczytu wynosi ok. 0,47 ns (wydajność 8,5 GB/s), dla skoku 128 bajtów ok. 0,54 ns (7,4 GB/s).

Istotny natomiast staje się przyrost wydajności przy zwiększeniu liczby generowanych żądań dostępu w pojedynczej iteracji pętli, uzyskany przez użycie wielu tablic. Dla zastosowanego kodu z czterema tablicami, wzrost wydajności w stosunku do pojedynczej tablicy wynosi ok. 4 (przy parametrach 0,14 ns, 29,5 GB/s) dla skalarnych rozkazów odczytu z pamięci i sięga prawie 30 (przy parametrach 0,018 ns, 221,5 GB/s) w przypadku zastosowania wektoryzacji kodu. Duża, ponad siedmiokrotna, różnica między wydajnością kodu wektorowego i kodu skalarnego jest cechą wyróżniającą odczyty z pamięci L1 w stosunku do innych poziomów pamięci.

Podobna jak powyżej analiza w przypadku pamięci L2 pokazuje, tak jak dla L1, brak wpływu po-

Poziom w hierarchii pamięci	L1		L2		L3		DRAM	
Parametr wydajnościowy	ns	GB/s	ns	GB/s	ns	GB/s	ns	GB/s
Organizacja odczytów z pamięci								
4 tablice, liniowe, rozkazy wektorowe	0,018	221,5	0,04	91,3	0,07	55,7	0,21	18,6
4 tablice, liniowe, rozkazy skalarne	0,14	29,5	0,16	25,3	0,16	25,8	0,23	17,1
1 tablica, liniowe (skok 4 bajty)	0,47	8,5	0,47	8,5	0,47	8,5	0,48	8,3
1 tablica, liniowe (skok 128 bajtów)	0,54	7,4	0,68	5,9	1,48	2,7	5,03	0,8
j.w. + *pointer chasing*	1,67	2,4	3,55	1,3	4,25	0,9	11,81	0,3
j.w. + losowa permutacja	1,67	2,4	3,53	1,1	9,32	0,4	61,73	0,06

Tablica 4.1: Wyniki wydajnościowe odczytów z pamięci dla jednowątkowego programu uruchomionego na rdzeniu o mikroarchitekturze Haswell mikroprocesora Intel Core i7-4790

bierania z wyprzedzeniem w przypadku *pointer chasing* (czas odczytu z permutacją i bez równy ok. 3,5 ns, przy wydajności ok. 1,2 GB/s) oraz następujące spadki czasu odczytu i przyrosty wydajności w kolejnych badanych przypadkach: dostęp linearny ze skokiem 128 bajtów – 0,68 ns, (5,9 GB/s) oraz ze skokiem 4 bajty – 0,47 ns (8,5 GB/s), wiele tablic z rozkazami skalarnymi – 0,16 ns (25,3 Gb/s), natomiast z rozkazami wektorowymi – 0,04 ns (91,3 GB/s). W stosunku do własności pamięci L1 zaznacza się większa różnica w wydajności miedzy dostępami linearnymi z różnym skokiem oraz mniejsza różnica w wydajności pomiędzy odczytami w kodzie skalarnym i wektorowym. Przy nieznacznie zmienionych proporcjach między poszczególnymi wariantami, ostateczny stosunek najszybszego do najwolniejszego czasu odczytu, a co za tym idzie wydajności, ponownie jak dla L1 przekracza 80.

W przypadku pamięci podręcznej poziomu L3 zaznaczają się ponownie różnice charakterystyk w stosunku do pamięci L1 i L2: maleje różnica pomiędzy wydajnością skalarną i wektorową (0,07 i 0,16 ns, 55,7 i 25,8 GB/s), rośnie różnica pomiędzy wydajnością dostępu linearnego ze skokiem 4 bajty i 128 bajtów (0,47 i 1,48 ns, 8,5 i 2,7 GB/s), pojawia się także różnica pomiędzy wydajnością w technice *pointer chasing* pomiędzy przypadkiem bez permutacji – 4,25 ns (0,9 GB/s) i z permutacją – 9,32 ns (0,4 GB/s), co wskazuje na odgrywanie roli przez *prefetching*. Ostateczny stosunek wydajności maksymalnej do minimalnej osiąga wartość ok. 140, znacząco większą niż dla pamięci L1 i L2.

Ostatnim analizowanym poziomem pamięci jest pamięć DRAM, często najważniejsza dla wydajności programu, często jedyna badana.

Pierwszą jej cechą, która rzuca się w oczy jest praktyczny brak różnicy w wydajności pomiędzy dostępami za pomocą rozkazów skalarnych i wektorowych (w przypadku kodu z odczytem z kilku tablic w pojedynczej iteracji) – odpowiednio 0,21 i 0,23 ns (18,6 i 17,1 GB/s) uzyskane w eksperymentach. Wskazuje to na znacznie niższe niż w przypadku pamięci podręcznych wymagania liczby generowanych współbieżnie żądań dostępu do pamięci potrzebnych do pełnego wykorzystania możliwości sprzętowych.

Mimo relatywnie małej liczby żądań odczytu wymaganych do wysycenia dostępnej przepustowości pamięci DRAM, okazuje się, że sposób organizacji dostępów może znacząco zwiększyć czas pojedynczego dostępu. Kolejne kilkukrotne wzrosty czasu odczytu dotyczą: dostępów z pojedynczą tablicą w iteracji – 0,48 ns (8,3 GB/s), dostępów ze skokiem 128 bajtów – 5,03 ns (0,8 GB/s), dostępów z uniemożliwionym przetwarzaniem potokowym rozkazów odczytu (*pointer chasing*) – 11,81 ns (0,3 GB/s) i ostatecznie z dodatkowo uniemożliwionym pobieraniem z wyprzedzeniem (brak *prefetchingu*) – 61,73 ns (0,06 GB/s).

Ostateczny stosunek najdłuższego czasu odczytu do najkrótszego dla pamięci DRAM przekracza 290, co jest wartością najwyższą ze wszystkich poziomów pamięci. Pokazuje to, jak wiele można zyskać lub stracić odpowiednio organizując dostępy do pamięci DRAM w programie.

Tabela 4.1 zawierająca, omówione powyżej, uzyskane eksperymentalnie uśrednione dane, podsumowuje badania wydajnościowe dotyczące hierarchii pamięci. Zawarte w niej liczby mogą stać się pomocą, przy projektowaniu i implementacji kodu, wskazując jakie jego własności, w szczególności dotyczące dostępów do pamięci podręcznych i pamięci DRAM, mają największe znaczenie dla uzyskiwanej wydajności kodu i jego czasu wykonania.

Ważną wartością, pokazującą skalę możliwych zysków i strat, jest stosunek najdłuższego czasu odczytu wartości pojedynczej zmiennej na badanej platformie sprzętowej (dostęp do pojedynczej tablicy w pamięci DRAM bez pobierania z wyprzedzeniem, bez potokowego przetwarzania, ze skokiem powodującym brak lokalności przestrzennej i małe wykorzystanie danych pobieranych w liniach pamięci podręcznej) do najkrótszego czasu odczytu (wektorowy dostęp do wielu tablic w pamięci L1 z pełnym przetwarzaniem potokowym) wynoszący kilka tysięcy (dokładniej ponad 3400). Stosunek ten powinien z pewnością, dla każdego programisty zainteresowanego czasem wykonania tworzonej aplikacji, stanowić fakt wart zastanowienia.

4.6 Szacowanie czasu dostępu do hierarchii pamięci w analizie wykonania programów

Szacowanie czasu realizacji operacji dostępu do danych (odczytu i zapisu) ma istotne znaczenie przy analizie czasów wykonania programów w świetle wspomnianego na początku tego rozdziału zjawiska ”memory wall” – rosnącej dysproporcji między wydajnością potoków przetwarzania a wydajnością pamięci DRAM. Przeprowadzone testy z użyciem odpowiednich benchmarków pozwalają na oszacowanie czasu wykonywania pojedynczej operacji arytmetycznej jako będącego rzędu ułamka nanosekundy (w rozważanych benchmarkach od ok. kilkunastu tysięcznych nanosekundy do ok. jednej nanosekundy), a czasu realizacji pojedynczego dostępu do pamięci DRAM jako rzędu nanosekund (w rozważanych benchmarkach od ok. dwóch dziesiątych nanosekundy do kilkudziesięciu nanosekund). Powyższe dane, uzyskane dla stosowanego w książce mikroprocesora i typowe dla większości współczesnych platform sprzętowych, prowadzą do wniosku o dominacji czasu dostępów do pamięci w całkowitym czasie wykonania programów, w których potoki pracują w sposób nieodbiegający znacząco od optymalnego, a dostępy do danych realizowane są z pamięci DRAM, przy czym liczba dostępów nie jest znacząco niższa od liczby operacji arytmetycznych.

Programy takie występują często, w wielu dziedzinach zastosowań. Szansą na ich optymalizację i skrócenie czasu wykonania jest maksymalizacja wykorzystania pamięci podręcznych, dla których czasy dostępów są istotnie krótsze od czasu dostępu do pamięci DRAM (tabela 4.1). W przypadku, kiedy udaje się realizować znaczącą liczbę dostępów do danych korzystając z kopii w pamięciach podręcznych, istotne staje się uwzględnienie tego faktu w szacowaniu czasu wykonania programów. Jednym z problemów przy tego typu analizie jest fakt, że w kodzie asemblera nie ma informacji o tym z jakiego zasobu pamięciowego korzysta dana operacja odczytu lub zapisu – kwestia ta zależy wyłącznie od charakterystyk sprzętu realizującego własne strategie organizacji dostępów do pamięci.

Analizując asembler napotyka się tylko rozkazy z adresami w pamięci wirtualnej, dla których zorientowanie się we wzorcu dostępu do pamięci może być utrudnione ze względu na konieczność śledzenia zawartości szeregu rejestrów używanych przy adresowaniu. Zazwyczaj łatwiejszym sposobem badania szczegółów realizacji dostępów do pamięci, w tym wykorzystania pamięci podręcznych, jest analiza kodu źródłowego, który często pozwala na prostą ilustracją struktury danych programu, a to w konsekwencji ułatwia ustalenie w jaki sposób realizowane są konkretne operacje dostępu do danych w programie.

Przy analizie wydajności programów, celem takich badań jest ustalenie, dla każdej operacji odczytu i zapisu, z jakiego poziomu w hierarchii pamięci korzysta operacja i jaka jest w związku z tym wydajność

realizacji tej operacji. Pierwszy z tych celów może polegać na ustaleniu liczby chybień w dostępie do pamięci podręcznej każdego z poziomów, przy założeniu, że każdy rozkaz dostępu prowadzi pierwotnie do próby znalezienia danej wartości (trafienia) w pamięci podręcznej najbliższej potokom przetwarzania (L1), a w przypadku chybienia podejmowana jest próba trafienia w pamięci L2, następnie dla kolejnego chybienia sprawdzana jest L3 i ostatecznie w przypadku trzech chybień (w L1, L2 i L3) dostęp realizowany jest z pamięci DRAM.

W podstawowym przyjętym w książce modelu, chybienie w pamięci konkretnego poziomu i brak chybienia w pamięci następnego poziomu oznacza transfer z pamięci dalszej od potoków przetwarzania z wydajnością charakterystyczną dla tego poziomu (transfer zawsze dotyczy całej linii pamięci podręcznej). Brak jakiegokolwiek chybienia oznacza transfer pojedynczej danej z L1, chybienie w pamięciach wszystkich poziomów powoduje dostęp do pamięci DRAM.

Model ten, jak wskazują rozważania niniejszego rozdziału, jest dla współczesnych mikroprocesorów tylko przybliżony. Rozważane w nim chybienia w pamięci podręcznej są do pewnego stopnia teoretyczne, ze względu na stosowaną w praktyce przez sprzęt technikę pobierania z wyprzedzeniem (*prefetching*). Wynikające z analizy wykonania kodu chybienie może nie nastąpić, kiedy odpowiednia linia pamięci podręcznej zostaje uprzednio dostarczona do pamięci dzięki mechanizmowi *prefetchingu*. Niemniej, każde takie teoretyczne chybienie oznacza jednak transfer danych – albo jako obsługę chybienia, albo jako wynikające z zastosowania *prefetchingu*.

W analizie wykonania kodu w ramach przyjętego w książce modelu, pozostawione zostaje powiązanie ustalonych na podstawie analizy teoretycznej chybień z transferami danych, zmienia się jednak powiązanie transferów z konkretnymi zdarzeniami sprzętowymi. Sprzętowe zdarzenia chybienia (L1_MISS, L2_MISS) nie uwzględniają działania pobierania z wyprzedzeniem i muszą zostać zastąpione innymi zdarzeniami bezpośrednio powiązanymi z transferami danych. Stąd w dalszych rozważaniach jako zdarzenia służące do szacowania transferu do pamięci L1 wykorzystywane są zdarzenia L1D_REPLACEMENT, a do obliczania transferu do L2 używane są zdarzenia L2_LINES_IN.ALL.

Praktycznie nierozwiązany pozostaje problem niezawodnego szacowania transferu danych z pamięci DRAM na podstawie zliczania zdarzeń sprzętowych. Pierwszą z trudności jest dobór takich zdarzeń. Pamięć podręczna L3 znajduje się poza rdzeniem i zliczanie faktów związanych z L3 (dostępy, chybienia) powinno być możliwe dzięki zdarzeniom z grupy OFFCORE_REQUESTS. Nie zawsze jednak udaje się uzyskać wiarygodne dane na temat tych zdarzeń. Alternatywnie można próbować zastosować zliczanie zdarzeń takich jak np. MEM_LOAD_UOPS_RETIRED.L3_MISS lub MEM_LOAD_UOPS_L3_MISS_RETIRED.LOCAL_DRAM, w praktyce jednak żadne z nich nie daje gwarancji poprawnego i jednoznacznego wskazania liczby dostępów do pamięci DRAM. W dalszych analizach w książce szacowanie transferu danych z pamięci DRAM przeprowadzane jest wyłącznie na podstawie rozważań teoretycznych i symulacji programem *valgrind* (p. 5.3.1).

Przedstawiony model szacowania transferów danych pomiędzy poziomami hierarchii pamięci może odbiegać od szczegółów funkcjonowania sprzętu także w innych aspektach. Rzeczywiste funkcjonowanie może być bardziej złożone, np. ze względu na specyfikę funkcjonowania pamięci podręcznych (przykładowo *inclusive cache* versus *exclusive cache*, a także utrzymywanie spójności pamięci podręcznych, *cache coherence*, omawiane w drugiej części książki). Działanie układu pamięci może być także niekiedy prostsze niż w modelu, jak w przypadku bezpośredniego dostępu do pamięci DRAM, kiedy pomija się wszystkie poziomy pamięci podręcznej.

Przy wszystkich omówionych powyżej ograniczeniach przedstawiony powyżej model używany jest w książce jako standardowy, stanowiąc kompromis między dokładnością odwzorowania pracy sprzętu i łatwością stosowania w analizach wydajności programów.

Rozdział 5

Optymalizacja klasyczna i kompilatory optymalizujące

Celem optymalizacji wydajności jest uzyskanie programu, który będzie dawał poprawne wyniki w możliwie najkrótszym czasie[1]. Jak było to już widoczne w przypadku wielu analizowanych wcześniej mikrobenchmarków, konieczność stosowania optymalizacji wynika z dużych różnic w czasach wykonania między programami nieoptymalnymi i zoptymalizowanymi.

Optymalizacja programu, będąca przedmiotem analiz w książce, polega na modyfikacjach kodu źródłowego, dokonywanych przez programistę oraz modyfikacjach rozmaitych pośrednich form zapisu programu i ostatecznej postaci asemblera, dokonywanych przez kompilatory. W pierwszym przypadku będziemy mówili o optymalizacji manualnej, w drugim o optymalizacji automatycznej.

Rozmaite techniki optymalizacji możemy podzielić na techniki niezależne od sprzętu, które powinny dawać pożądany efekt dla standardowych środowisk wykonania, oraz techniki ukierunkowane na wykorzystanie specjalnych możliwości konkretnych mikroprocesorów lub rodzin mikroprocesorów.

Ze stosowaniem optymalizacji zawsze wiąże się problem przenośności. Optymalizacje ukierunkowane na specyficzne cechy sprzętu mogą prowadzić do nieprzenośnego kodu. Co więcej, także optymalizacje ogólnego przeznaczenia mogą stanowić problem. Zazwyczaj gwarantują one możliwość kompilacji i uruchomienia w dowolnym środowisku tworzenia i wykonania oprogramowania oraz poprawność programów zoptymalizowanych, jednak poza tak rozumianą przenośnością, można także rozważać przenośność wydajności – czy kod poddany optymalizacji będzie zawsze prowadził do skrócenia czasu wykonania, niezależnie od kompilatora i środowiska wykonania?

Kolejnym z pojawiających się problemów, jest współgranie optymalizacji z innymi jeszcze, poza przenośnością, celami związanymi z tworzeniem programów: łatwością i czasem pisania kodu źródłowego, a następnie łatwością utrzymania i dalszego rozwijania programu[2]. Cele te i środki do nich prowadzące mogą być wzajemnie sprzeczne, a ostateczny wybór strategii rozwijania oprogramowania może zależeć od wielu czynników.

Techniki optymalizacji można podzielić na tzw. optymalizacje klasyczne, znane i stosowane od wielu lat dla programów jednowątkowych, oraz techniki związane z rozwojem sprzętu w ostatnich latach: wektoryzację i zrównoleglenie. W niniejszym rozdziale omówione są wybrane techniki optymalizacji klasycznej. Zakładane jest także możliwe dokonywanie wektoryzacji kodu przez kompilatory, polegające na użyciu rejestrów i rozkazów wektorowych. Zrównoleglenie programów analizowane jest w dalszej części książki.

[1] Zgodnie z założeniami przyjętymi w książce, optymalizacja ze względu na czas wykonania jest jedyną badaną, stąd określenie optymalizacja odnosi się zawsze do niej.

[2] Do trudności z wkomponowaniem optymalizacji w proces tworzenia oprogramowania nawiązuje słynne stwierdzenie Donalda Knutha: "premature optimization is the root of all evil".

5.1 Optymalizacja klasyczna

Optymalizacje, manualna i automatyczna, stosują szereg technik, których istota jest często niezależna od etapu tworzenia programu wykonywalnego i sposobu zapisu kodu. Aby osiągnąć cel optymalizacji, skrócenie czasu wykonania programu, dąży się najczęściej do:

- zmniejszenia liczby użytych rozkazów asemblera
- stosowania bardziej wydajnych rozkazów (umożliwiających realizację większej liczby operacji w jednostce czasu lub określonej liczbie taktów procesora)
- usuwania zależności pomiędzy rozkazami
- optymalnego wykorzystania hierarchii pamięci

Przegląd technik optymalizacji klasycznej rozpoczynają proste modyfikacje kodu, stosowane standardowo przez kompilatory, a więc rzadziej wymagane na etapie tworzenia kodu źródłowego. Zapis przykładowych zastosowań przedstawiony jest jednak w języku C, intuicyjnie łatwiejszym do interpretacji od asemblera. W przypadku optymalizacji kodu asemblera szczegóły realizacji mogą się różnić (np. zmienne zastępowane są przez rejestry), jednak idea optymalizacji pozostaje niezmienna. Nazwy technik podawane są w języku angielskim, w literaturze polskiej brak jest ugruntowanych tłumaczeń nazw.

constant folding – optymalizacja polegająca na zastąpieniu pojawiającego się wielokrotnie wyrażenia zawierającego stałe (wymagającego każdorazowo wykonania pewnych operacji) przez nową stałą, równą obliczonej jednokrotnie wartości wyrażenia (co eliminuje wielokrotne wykonywanie operacji):

- kod przed optymalizacją (z symbolicznie zaznaczoną pętlą wskazującą na wielokrotne wykonywanie operacji zawartej w pojedynczej iteracji):

  ```
  for(i=...) o = 2*PI*r[i];
  ```

- kod po optymalizacji (redukcja jednego mnożenia w każdej iteracji):

  ```
  const double 2_PI = 2*PI;  for(i=...) o = 2_PI*r[i];
  ```

copy propagation – zastąpienie zmiennych (rejestrów) zawierających kopie wartości pewnej pierwotnej zmiennej (rejestru) przez tę zmienną (rejestr); celem użycia pierwotnej zmiennej jest usunięcie ewentualnych zależności (patrz p. 3.4.4) związanych ze zmienną będącą kopią:

- kod przed optymalizacją (zawiera zależność rzeczywistą, *read-after-write*, ze względu na zmienną *y*, która jest początkowo zapisywana jako kopia *x*, a następnie odczytywana, co uniemożliwia współbieżne wykonanie obu operacji) :

  ```
  y = x; ...; z = f(y);
  ```

- kod po optymalizacji (bez zależności, występują tylko odczyty zmiennej x, możliwe do współbieżnej realizacji):

  ```
  y = x; ...; z = f(x);
  ```

strength reduction – wykorzystanie instrukcji (rozkazów) uzyskujących zamierzonych efekt w sposób bardziej wydajny:

- kod przed optymalizacją (w podanym przykładzie założone jest, że implementacja funkcji potęgowania `pow` zawsze korzysta z iteracyjnego algorytmu odpowiedniego dla liczb podwójnej precyzji, bez optymalizacji dla wykładników całkowitych):

  ```
  y = pow(x,4);
  ```

- kod po optymalizacji (dwie operacje zamiast całego algorytmu):

  ```
  temp = x*x; y = temp*temp;
  ```

common subexpression elimination (CSE) – powszechnie stosowana optymalizacja polegająca na zamianie pewnego powtarzającego się wielokrotnie w kodzie źródłowym wyrażenia przez wartość zmiennej przechowującej jednokrotnie obliczoną jego wartość (podobnie jak *constant folding* redukuje liczbę operacji wykonywanych w kodzie, dotyczy jednak wyrażeń zawierających zmienne, nie wyłącznie stałe)

- kod przed optymalizacją:

  ```
  a = b * c + g; d = b * c * e;
  ```

- kod po optymalizacji:

  ```
  temp = b*c; a = temp + g; d = temp * e;
  ```

Spośród optymalizacji klasycznych szczególnie istotne są optymalizacje związane z realizacją pętli (w praktyce najczęściej o czasie wykonania całego kodu decyduje czas realizacji występujących w nim pętli o dużej liczbie iteracji). Poniżej wymienione jest kilka przykładowych optymalizacji, o różnej częstości stosowania w praktyce i różnym możliwym wpływie na czas wykonania.

loop invariant code motion (LICM) – często stosowana optymalizacja, niekiedy będąca rozszerzeniem optymalizacji CSE – operacje wielokrotnie powtarzane w pętli są umieszczane przed pętlą, a w pętli znajduje się wyłącznie ich wcześniej obliczony wynik:

- kod przed optymalizacją – podwójna pętla z dostępem do macierzy (tablicy 2D) przechowywanej wierszami w jednowymiarowej tablicy `a` (patrz. p. 2.1.1), z indeksem w tablicy `a` obliczanym na podstawie indeksów 2D:

  ```
  for(i=0; i<N; i++){
    for(j=0; j<N; j++) {
      sum += a[i*N+j];
  }}
  ```

- kod po optymalizacji:

  ```
  for(i=0; i<N; i++){
    int in = i*N;
    for(j=0; j<N; j++) {
      sum += a[in+j];
  }}
  ```

induction variable simplification (IVS) – pod tą nazwą kryją się różne optymalizacje dotyczące zmiennych, których wartości modyfikowane są o stałą wartość w każdej iteracji pętli (wartości te są często liniową funkcją indeksu pętli); np. w przypadku rozważanych uprzednio pętli z dostępami do tablicy, IVS prowadzi do redukcji liczby wykonywanych operacji i uniezależnienia obliczania indeksu w tablicy od indeksów obu pętli za pomocą poniższych modyfikacji:

- kod przed optymalizacją:

```
for(i=0; i<N; i++){
  for(j=0; j<N; j++) {
    sum += a[i*N+j];
}}
```

- kod po optymalizacji (w stosunku do optymalizacji LICM dodatkowym zyskiem jest zastąpienie dodawania przy obliczaniu indeksu w tablicy inkrementacją, będącą z założenia operacją efektywniejszą):

```
int in = 0;
for(i=0; i<N; i++){
  for(j=0; j<N; j++) {
    sum += a[in]; in++;
}}
```

loop fusion – oczywistą optymalizacją bywa połączenie dwóch pętli o identycznych zakresach indeksów w jedną pętlę, mogące prowadzić, poza redukcją liczby operacji związanych z modyfikacją indeksu pętli i obliczaniem wyrażenia determinującego zakończenie pętli, także do innych optymalizacji:

- kod przed optymalizacją:

```
for(i=0; i<N; i++){
    b[i] += a[i];
}
for(i=0; i<N; i++){
    d[i] = c[i]*b[i];
}
```

- kod po optymalizacji – dodatkowym uzyskanym efektem jest redukcja liczby dostępów do tablicy `b` (zakładając odpowiednie wykorzystanie rejestru z tymczasową wartością `b[i]`):

```
for(i=0; i<N; i++){
    b[i] += a[i];
    d[i] = c[i]*b[i];
}
```

loop fission – jest odwrotną do *loop fusion* optymalizacją, polegającą na rozdzieleniu jednej pętli na dwie lub więcej mniejszych; przykład takiej optymalizacji, służącej redukcji tzw. ciśnienia na rejestry (*register pressure*), omówiony jest w p. 3.10.1

loop interchange – zamiana kolejności wykonywania zagnieżdżonych pętli najczęściej zmierza do optymalizacji dostępów do pamięci:

- kod przed optymalizacją – dostęp do macierzy 2D przechowywanej wierszami w tablicy jednowymiarowej, z pętlą po kolumnach jako zewnętrzną (w każdej kolejnej iteracji pętli wewnętrznej dostęp do elementu tablicy `a` oddalonego o `N` wyrazów od elementu pobranego w iteracji poprzedniej):

```
for(j=0; j<N; j++){
  for(i=0; i<N; i++){
    sum += a[i*N+j];
}}
```

- kod po optymalizacji – oczywiste poprawienie lokalności przestrzennej poprzez ustalenie pętli po wierszach jako pętli zewnętrznej, a pętli po kolumnach, czyli kolejnych elementach w każdym wierszu, jako wewnętrznej (w każdej iteracji pętli wewnętrznej odczytywana jest wartość elementu tablicy położonego w pamięci bezpośrednio po elemencie pobranym w iteracji poprzedniej):

```
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    sum += a[i*N+j];
}}
```

loop unrolling – rozwinięcie pętli, o zaletach którego była już mowa kilkakrotnie wcześniej, polega na złożonej transformacji, w której rozbija się oryginalną pętlę, o np. N iteracjach, na dwie odrębne pętle. W pierwszej z nich realizowana jest zdecydowana większość operacji, przy czym w pojedynczej iteracji wykonuje się treść kilku (np. k) iteracji pętli oryginalnej, a liczba iteracji wynosi N/k (liczbę k nazywa się czynnikiem rozwinięcia pętli). Powoduje to automatycznie zmniejszenie liczby operacji modyfikacji indeksu pętli i porównania go z graniczna wartością (ewentualnie innych operacji zawartych w wyrażeniach instrukcji `for`), a jednocześnie (być może po dalszych transformacjach) umożliwia np. wektoryzację kodu (z tej przyczyny czynnik rozwinięcia pętli k jest często potęgą 2, np. 4 lub 8). W drugiej utworzonej pętli wykonuje się kilka iteracji, tak aby ostateczny wynik po optymalizacji był identyczny z wynikiem pętli oryginalnej (liczbę tych iteracji oblicza się jako resztę z dzielenia oryginalnej liczby iteracji przez czynnik rozwinięcia pętli, w założonym przykładzie jest to $N\%k$):

- kod przed optymalizacją (jako przykład użyta jest pętla służąca do obliczenia kwadratu normy wektora przechowywanego w tablicy `X`):

```
for(i=0; i<N; i++){
 norm2 += X[i]*X[i];
}
```

- kod po optymalizacji – powyższa pętla rozwinięta o czynnik 4 (przykład pomija drugą pętle powstałą w ramach transformacji *loop unrolling*, w której wykonywane jest `N`%4 pierwotnych iteracji):

```
for(i=0; i<N; i+=4){
 norm2 += X[i]*X[i]+X[i+1]*X[i+1]+X[i+2]*X[i+2]+X[i+3]*X[i+3];
}
```

register blocking, cache blocking – obie optymalizacje związane są z rozwinięciami wielokrotnie zagnieżdżonych pętli, a ich omówienie znajduje się w dalszych punktach, poświęconych bardziej rozbudowanym przykładom optymalizacji kodu.

Dotychczas omówione optymalizacje, dotyczące wyrażeń i prostych manipulacji indeksami pętli oraz wyrażeniami od nich zależnymi, nie wyczerpują zestawu technik stosowanych przez kompilatory i możliwych także do użycia przez programistów. Poniżej znajduje się kilka innych ważnych lub ciekawych optymalizacji:

dead code removal – optymalizacja polegająca na usunięciu (pominięciu podczas kompilacji) kodu, który nie produkuje wyniku wykorzystywanego w dalszej części programu. Przykładami takiego kodu mogą być np. rozmaite warianty benchmarków mierzących wydajność. Jeżeli kompilator na podstawie analizy kodu ustali, że efekt realizacji benchmarku nie jest dalej użyty, może usunąć z programu cały kod benchmarku, prowadząc do błędnych pomiarów wydajnościowych. W celu przeciwdziałania takim sytuacjom, w przykładach wykorzystywanych w książce stosowane są pewne wybrane proste operacje, spośród szeregu możliwych działań związanych z wynikami obliczeń, takich jak wypisywanie na ekranie, zapisywanie do pliku, ewentualnie przekazanie jako argumenty do dowolnej, często sztucznie utworzonej, funkcji. Istotnym w takich przypadkach jest niedopuszczenie do zaburzenia wyników wydajnościowych przez operacje przeciwdziałające usuwaniu kodu. W poniższym przykładzie badana jest wydajność dodawania dwóch wektorów o długości *N*, powtarzana wielokrotnie (*NTIMES* razy) w celu uzyskania odpowiednio długich czasów wykonania, gwarantujących dokładność pomiaru (wektory mogą być krótkie, np. wtedy kiedy mierzone są charakterystyki wydajnościowe pamięci podręcznych):

- kod niezabezpieczony przed usunięciem wielokrotnego powtórzenia dodawania wektorów – w każdej iteracji pętli zewnętrznej realizowane są identyczne operacje w pętli wewnętrznej, kompilator może całkowicie usunąć pętlę zewnętrzną i pozostawić tylko jedno wykonanie dodawania wektorów w pętli wewnętrznej, a więc jedną iteracje pętli zewnętrznej, wynik pozostałych, na pewno nie odgrywa żadnej roli w kodzie:

```
for (k=0; k<NTIMES; k++) {
  for (j=0; j<N; j++) a[j] = b[j] + c[j];
}
```

- kod zabezpieczony przed optymalizacją *dead code removal*:

```
for (k=0; k<NTIMES; k++) {
  for (j=0; j<N; j++) a[j] = b[j] + c[j];
  j=N-1; tmp = b[j]+c[j]*a[j];
  if (tmp < -1.0e-4) tmp += f_never(a,b,c,N);
}
```

  Z danych wykorzystywanych w programie wynika, że warunek dla zmiennej *tmp* nigdy nie jest spełniony, a więc wywołanie funkcji `f_never`, które mogłoby stanowić narzut wydajnościowy, nigdy nie następuje i, poza obliczeniem wartości zmiennej `tmp` oraz sprawdzeniem warunku, wykonywany kod pozostaje identyczny z kodem sprzed optymalizacji. Kompilator mógłby w związku z tym usunąć pętlę zewnętrzną i pozostawić tylko jednokrotne obliczanie sumy wektorów, jednak nie jest w stanie wywnioskować z kodu, że funkcja `f_never` nie jest wykonywana i będzie musiał pozostawić wielokrotne wykonanie iteracji pętli zewnętrznej (fakt istnienia wywołania funkcji, której argumentami są wektory, powoduje, że elementy wektorów mogą być zmieniane wewnątrz funkcji, produkując różne wyniki w każdej iteracji pętli zewnętrznej, w tym różne wartości zmiennej *tmp*). Do uniemożliwienia usunięcia kodu potrzebne jest jeszcze użycie, np. wypisanie na ekranie, w dalszej części kodu, wartości zmiennej *tmp*. Charakter dodatkowych operacji w pętli zewnętrznej oraz fakt, że dodatkowa funkcja nie jest nigdy wywoływana, sprawiają, że pomiar wydajnościowy w benchmarku nie powinien być znacząco zaburzony.

inlining – wplatanie kodu funkcji w miejscu wywołania, najczęściej realizowane jako optymalizacja automatyczna (często po użyciu odpowiednich opcji kompilacji), powoduje usunięcie narzutu czasowego związanego z obsługą wywołania, w tym operacji przekazywania argumentów i wyniku

funkcji. Prosty poniższy przykład nie wyczerpuje możliwości współczesnych kompilatorów dokonywania wplatania treści bardziej rozbudowanych i złożonych funkcji:

- kod przed optymalizacją:

```
double f1(double a, double b){ return(a+b); }
...
z = f1(x, y);
```

- kod po optymalizacji:

```
z = x + y;
```

software prefetching i software pipelining – realizację na poziomie kodu źródłowego sprzętowych optymalizacji pobierania z wyprzedzeniem (*prefetching*) i przetwarzania potokowego (*pipelining*) można czytelnie zilustrować na przykładzie pętli przetwarzania zawartości kolejnych struktur przechowywanych w tablicy `str_tab` (przetwarzanie pojedynczej struktury dokonywane jest wewnątrz funkcji *process*):

- kod przed optymalizacją (w każdej iteracji układ realizujący przetwarzanie zawartości struktury czeka na dostarczenie danych z pamięci):

```
for(i=0; i<N; i++){
  process( str_tab[i] );
}
```

- kod po optymalizacji – operacje na pojedynczej strukturze są rozbite na dwa etapy: pobierania (*fetch*) i przetwarzania (*process*), a pętla zorganizowana jest w taki sposób, że w pojedynczej iteracji wykonywane są obie operacje, ale na dwóch kolejnych strukturach tablicy. W oczywisty sposób nawiązuje to do optymalizacji sprzętowych, gdzie pojedyncza iteracja odpowiada taktowi zegara (ewentualnie pewnej liczbie taktów), a operacje pobierania i przetwarzania wykonywane są współbieżnie przez odrębne układy sprzętowe:

```
fetch( str_tab[0] );
for(i=0; i<N-1; i++){
  fetch( str_tab[i+1] );
  process( str_tab[i] );
}
process( str_tab[N-1] );
```

Po optymalizacji, w każdej iteracji dane dla układu przetwarzania, dzięki wywołaniu *fetch* w poprzedniej iteracji, są już dostarczone albo znajdują się w drodze z pamięci.

W praktyce często stosuje się tzw. podwójne buforowanie. Zamiast wywołania imitującej pobieranie z pamięci funkcji *fetch*, zapisuje się pobieraną strukturę w zmiennej tymczasowej. W celu efektywnego zastosowania przetwarzania potokowego potrzebne są dwie takie zmienne (dwa bufory na dane):

```
str_tmp1 = str_tab[0];
for(i=0; i<N-1; i++){
  str_tmp2 = str_tab[i+1];
  process( str_tmp1 );
  str_tmp1 = str_tmp2;
}
process( str_tmp1 );
```

5.2 Kompilatory optymalizujące

Wszystkie dotychczas omawiane w książce programy uzyskiwane były poprzez kompilację kodu źródłowego, dokonywaną przez kompilatory uruchamiane z rozmaitymi opcjami optymalizacji. Optymalizacje te mają kluczowe znaczenie dla ostatecznej wydajności kodu, stąd w niniejszym rozdziale omówione są pewne podstawowe fakty dotyczące kompilatorów i procesu kompilacji.

Optymalizacje stosowane przez kompilatory polegają na takim doborze rozkazów procesora, aby skrócić czas wykonania programu, ewentualnie także zmniejszyć liczbę rozkazów i wielkość pamięci zajmowanej przez program po kompilacji oraz w trakcie wykonania (te ostatnie optymalizacje nie są w książce szczegółowo omawiane).

Pierwotnym kodem poddawanym optymalizacji przez kompilator jest forma programu uzyskana poprzez analizę leksykalną, składniową i semantyczną. Na różnych etapach kompilacji stosowane są różne rodzaje notacji, prowadzące do ostatecznego zapisu w języku asemblera. W dalszym ciągu jedynym badanym efektem kompilacji będzie kod asemblera, porównywane będą jego różne warianty, uzyskiwane po zastosowaniu różnych technik i opcji optymalizacji. Punktem wyjścia będzie kod otrzymany bez optymalizacji, naśladujący wiernie treść kodu źródłowego.

Podobnie jak w dotychczasowych przykładach interesować nas będą tylko krótkie fragmenty asemblera, o których, na podstawie wcześniejszej analizy kodu źródłowego lub badania wykonania programu, zakładamy, że mają decydujące znaczenie dla czasu wykonania.

W badaniu takich fragmentów pomocnym pojęciem jest pojęcie bloku podstawowego (*basic block*). Nazywana jest tak część kodu (zapisanego w dowolnej notacji, choć nas interesuje głównie język asemblera), złożona z sekwencji linii, charakteryzująca się:

- pojedynczym punktem wejścia (co oznacza, że żadna linia wewnątrz bloku nie jest celem rozkazu skoku),
- pojedynczym punktem wyjścia, z którego następuje przejście do innego bloku (co oznacza, że z założenia wszystkie linie w bloku podstawowym są wykonywane zawsze w kolejności zapisu, od pierwszej do ostatniej).

Ze względu na swoje cechy, bloki podstawowe nadają się dobrze do optymalizacji, a także do badania przez twórców oprogramowania. Łatwo jest powiązać blok podstawowy z sekwencją instrukcji w kodzie źródłowym, zanalizować optymalizacje zastosowane przez kompilator i rozważyć inne możliwe modyfikacje kodu. Wszystkie fragmenty kodu asemblera rozważane dotychczas w książce obejmowały tylko pojedyncze bloki podstawowe.

W ramach bloków podstawowych asemblera, realizując zapis tego samego fragmentu kodu źródłowego, kompilatory mogą stosować rozmaite rozkazy, w dowolnej kolejności, pod warunkiem produkowania zawsze poprawnych wyników przy każdym wykonaniu kodu. Sytuacja jest tutaj podobna do działania rdzeni mikroprocesora, które mogą stosować własne optymalizacje, np. rozbijać rozkazy asemblera na sekwencje mikrorozkazów, wykorzystywać wykonywanie poza kolejnością (*out of order execution*), wykonanie spekulatywne (*speculative execution*), itp.

Ilustracją działania kompilatorów optymalizujących jest poniższy przykład generowania kodu asemblera dla prostej pętli:

```
while( j < n ) {
  k += 2j;
  m = 2j;
  j++;
}
```

Wersja asemblera utworzona przez kompilator *gcc* bez opcji optymalizacji wygląda następująco:

```
.L2
   movl -4(%ebp), %eax      # j -> eax
   cmpl -12(%ebp), %eax     # n <> eax ?
   jl .L4
   jmp .L3
.L4
   movl -4(%ebp), %eax      # j -> eax
   movl %eax, %edx          # j -> edx
   leal 0(,%edx,2), %eax    # eax = 2*edx
   addl %eax, -8(%ebp)      # k += eax ( k += 2*j )
   movl -4(%ebp), %eax      # j -> eax
   movl %eax, %edx          # j -> edx
   leal 0(,%edx,2), %eax    # eax = 2*edx
   movl %eax, -16(%ebp)     # m = eax ( m = 2*j )
   incl -4(%ebp)            # j++
   jmp .L2
.L3
```

Komentarze pokazują, którym operacjom kodu źródłowego odpowiadają kolejne linie asemblera (wykorzystując nazwy zmiennych i rejestrów, standardowe symbole operacji arytmetycznych, operację podstawienia wartości zmiennej dla oznaczenia zapisu do pamięci oraz symbol -> dla odczytu z pamięci). Kompilator zastosował rozbicie operacji związanych z instrukcją *while* oraz operacji iteracji pętli na dwa osobne bloki podstawowe. Spełnienie warunku kontynuacji pętli, *j*<*n*, testowane w pierwszym bloku podstawowym prowadzi do przejścia do drugiego bloku podstawowego, realizującego treść iteracji, zaczynającego się od etykiety `L4`. Niespełnienie warunku powoduje przejście do dalszej części kodu, zaczynającej się od etykiety `L3`.

W kodzie asemblera zwraca uwagę fakt powtarzania operacji pobierania z pamięci wartości zmiennej *j* (i związane z tym powtarzanie wykonywania operacji mnożenia *2*j*), a także wykorzystanie operacji *leal* (*load effective address*, obliczanie adresu w złożonym trybie adresowania, bez wykonywania operacji na pamięci) do wykonywania mnożenia przez 2.

Wielokrotne pobieranie z pamięci i zapis do pamięci wartości zmiennych jest cechą charakterystyczną niezoptymalizowanego kodu asemblera, w szczególności przy jawnie zaznaczonej opcji produkowania kodu przeznaczonego do debugowania (po odpowiednich linijkach asemblera zawartość komórek pamięci przechowujących zmienne powinna być taka sama jak wynikająca z treści odpowiadających instrukcji kodu źródłowego). Takie zachowanie pozwala na debugowanie kodu źródłowego linijka po linijce.

Zastosowanie opcji optymalizacji powoduje redukcję liczby pobrań z pamięci i wykorzystanie rejestrów nie tylko do przechowywania wartości zmiennych w pojedynczej iteracji pętli (z odczytem do rejestrów i zapisem do pamięci w każdej iteracji), ale także przechowania tych wartości przez czas wykonania całej pętli (z wykorzystywaniem zawartości rejestrów we wszystkich iteracjach, bez odniesień do pamięci):

```
.L4
   leal (%edx, %eax, 2), %edx    # edx += 2*eax
   leal 0(,%eax,2), %ecx         # ecx = 2*eax
   incl %eax                     # eax += 1
   cmpl %ebx, %eax               # n <> eax ?
```

```
jl .L4
```

W powyższej wersji rejestr `eax` przechowuje na stałe wartość zmiennej *j*, a rejestry `ebx`, `ecx` i `edx` wartości zmiennych, odpowiednio: *n*, *m* i *k*. W oczywisty sposób kod taki nie pozwala na sprawdzenie zawartości komórek pamięci po każdej iteracji, jego wykonanie jednak będzie wielokrotnie szybsze niż kodu niezoptymalizowanego, przy jednoczesnym zachowaniu poprawności końcowego rezultatu.

Kolejna zoptymalizowana wersja asemblera dla przykładowej pętli pokazuje użycie optymalizacji IVS (*induction variable simplification*):

```
.L4:
   addl $1, %ecx          // j++
   addl %eax, %edx        // k+=m
   addl $2, %eax          // m+=2
   cmpl %ebx, %ecx        // n<>j ?
   jne .L4
```

Kompilator wykorzystał fakt, że w każdej iteracji pętli wartość zmiennej *m* zwiększa się o 2, co pozwoliło na zastosowanie optymalizacji IVS i uwolnienie operacji na zmiennych *k* i *m* od wykorzystywania zmiennej sterującej pętli *j*. Warto podkreślić, że kolejność wykonywania operacji w kodzie asemblera jest zupełnie inna niż w oryginalnym kodzie źródłowym.

5.2.1 Przykładowe opcje optymalizacji automatycznej

Przedstawiony powyżej pobieżny przegląd funkcjonowania współczesnych kompilatorów warto zakończyć uwagą o dziesiątkach innych opcji optymalizacji możliwych do wykorzystania. Stosowany najczęściej w książce kompilator *gcc* posiada ponad 200 opcji optymalizacji ogólnego przeznaczenia, a ponadto dodatkowe opcje dla wielu specyficznych architektur komputerowych. Niektóre z opcji odpowiadają klasycznym optymalizacjom omawianym w poprzednich punktach (np. *-funroll-loops, -fsplit-loops, -fmove-loop-invariants, -floop-interchange, -finline-functions*).

W przykładach opisywanych w książce, do kompilacji z reguły nie są wykorzystywane konkretne specyficzne opcje optymalizacji, takie jak wymienione powyżej. Stosowane są natomiast popularne zbiorcze opcje oznaczające całe zestawy optymalizacji, grupowane w tzw. poziomy optymalizacji. W standardowych kompilatorach najczęściej stosuje się cztery podstawowe poziomy optymalizacji (oznaczane odpowiednio *-O0, -O1, -O2, -O3*) oraz ewentualnie dodatkowe poziomy np. dla automatycznego zrównoleglenia kodu.

Kolejne poziomy optymalizacji najczęściej oznaczają:

-O0 - brak optymalizacji, skutkujący utworzeniem kodu do precyzyjnego debugowania

-O1 - podstawowe optymalizacje, zmierzające do redukcji czasu wykonania i rozmiaru kodu wyjściowego (binarnego)

-O2 - zaawansowane optymalizacje ukierunkowane na zwiększenie wydajności, nie powodujące pojawienia się ewentualnych błędów lub niedokładności przy wykonaniu programu

-O3 - agresywna optymalizacja, często związana z możliwością produkowania kodu nie odpowiadającego wszystkim wymaganiom standardów (np. może zawierać szczegółową opcję, oznaczaną często jako *-fast-math*, która stosuje szybkie obliczenia matematyczne, nie zawsze zgodne z wymaganiami standardów IEEE i ISO)

Konkretne kompilatory mogą definiować różne zestawy opcji dla różnych poziomów optymalizacji (np. *gcc* stosuje agresywne optymalizacje dopiero dla specjalnej opcji *-Ofast*). Zawsze jednak obowiązuje zasada, że wyższy poziom oznacza dalej idące optymalizacje, a więc i modyfikacje kodu. Szczegółowe dane zawarte są zawsze w dokumentacji kompilatorów, w systemach Unixowych zazwyczaj możliwe jest też wykorzystanie zainstalowanych podręczników (*manual*), np. dla kompilatora *gcc* za pomocą polecenia *man gcc*.

W książce przyjęte jest stosowanie odpowiednich poziomów optymalizacji, połączone z weryfikacją otrzymanego kodu asemblera. Zdarza się, że pewne poziomy optymalizacji nie wywołują oczekiwanego rezultatu, który można przewidzieć jako optymalny dla danego kodu źródłowego. Koniecznym jest wtedy zastosowanie konkretnych szczegółowych opcji, ewentualnie ręczne korygowanie kodu asemblera.

Poza użyciem poziomów optymalizacji, w książce stosowana jest często jedna konkretna opcja, polegająca na wskazaniu konkretnej architektury mikroprocesora, dla której należy dokonywać optymalizacji. W przypadku wykorzystywanych w książce jako przykładowy sprzęt mikroprocesorów firmy Intel, opcją tą jest *-march=core-avx2*, używana w celu wymuszeniu wektoryzacji kodu wykonywalnego.

5.3 Przykładowe optymalizacje dla wybranych algorytmów numerycznej algebry liniowej

5.3.1 Mnożenie macierz-wektor

Mnożenie macierz-wektor (*matrix-vector multiplication*) jest jedną z najczęściej wykorzystywanych w numerycznej algebrze liniowej operacji (jest np. podstawową operacją w algorytmach iteracyjnego rozwiązywania układów równań liniowych, a także rozmaitych algorytmach optymalizacji).

Matematycznie operacja mnożenia macierz-wektor, dla macierzy $\boldsymbol{A}$ o rozmiarze NxN i wektora $\boldsymbol{x}$, produkująca wektor $\boldsymbol{y}$, zapisywana jest jako:

$$\boldsymbol{y}_i = \sum_{j=0}^{N-1} \boldsymbol{A}_{ij}\boldsymbol{x}_j$$

We wzorze zastosowano indeksowanie od 0, typowe dla implementacji w języku C. Pojedynczy wyraz wektora $\boldsymbol{y}$ jest uzyskiwany jako iloczyn skalarny wiersza macierzy $\boldsymbol{A}$ i wektora $\boldsymbol{x}$. Uogólnienie na przypadek macierzy prostokątnej jest oczywiste i nie jest dalej szczegółowo analizowane.

Naiwną implementacją algorytmu, przy stosowanym w książce sposobie przechowywania macierzy za pomocą tablic jednowymiarowych, jest podwójna pętla:

```
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    y[i] += a[i*N+j]*x[j];
}}
```

(implementacja zakłada przechowywanie macierzy $\boldsymbol{A}$ w tablicy `a` oraz wcześniejsze zainicjowanie wektora `y` zerami, co jest, jak wykazują dalsze badania, operacją o czasie możliwym do pominięcia przy analizach wydajności).

Liczba operacji algorytmu wynosi N^2 mnożeń i tyle samo dodawań, co daje złożoność rzędu N^2. Z punktu widzenia czasu wykonania istotny jest także czas dostępów do pamięci. Najmniej wątpliwości budzą dostępy do tablicy `a` – algorytm wymaga odczytania wszystkich N^2 elementów macierzy, każdego jednokrotnie.

Wszystkie iteracje pętli wewnętrznej dotyczą pojedynczego elementu wektora `y`. Odczytywanie i zapisywanie tego elementu w pamięci w każdej iteracji może mieć sens tylko w przypadku precyzyjnego

debugowania, natomiast dla wersji ukierunkowanej na maksymalizację wydajności jedynym sensownym rozwiązaniem jest przechowywanie aktualizowanej wartości elementu wektora `y` w rejestrze przez wszystkie iteracje pętli wewnętrznej. Na poziomie kodu źródłowego można próbować uzyskać taki efekt przez wprowadzenie zmiennej tymczasowej `y_i` (wbrew intuicyjnej oczywistości, nie wszystkie kompilatory stosują modyfikację, polegającą na użyciu pojedynczego rejestru dla tymczasowych wartości elementu `y[i]`, nawet dla wysokich poziomów optymalizacji *-O2*, *-O3*):

```
for(i=0; i<N; i++){
  y_i = 0.0;
  for(j=0; j<N; j++){
    y_i += a[i*N+j]*x[j];
  }
  y[i] = y_i;
}
```

Zastosowana zmiana powoduje redukcję, na poziomie kodu źródłowego i w konsekwencji także podczas wykonania, liczby dostępów do elementów wektora `y` do wyłącznie *N* zapisów. Zakładając odpowiednio dużą wartość *N*, oznacza to czas pomijalny w stosunku do czasu dostępów do elementów tablicy `a` (w analizach algorytmów zawsze czasy rzędu będącego niższą potęgą *N* będą pomijane w stosunku do czasów odpowiadających wyższej potędze *N*).

Ostatnim z czynników wpływających na czas wykonania mnożenia macierz-wektor jest czas dostępów do wektora `x`. Podstawowa, naiwna analiza wydajności opiera się bezpośrednio na kodzie źródłowym. W każdej iteracji pętli wewnętrznej znajduje się dostęp do kolejnego elementu wektora `x`, co można uznać za odpowiadające pobraniu z pamięci. Czas dostępów jest więc czasem N^2 odczytów.

Bardziej szczegółowe badanie wydajności prowadzi do wniosku, że w analizowanej implementacji czas dostępów do wektora `x` zależy od rozmiarów pamięci podręcznych różnych poziomów oraz od wymiaru *N*. Dla dużych pamięci podręcznych i małej wartości *N*, wektor `x` mieści się w całości w najmniejszej z pamięci *cache*, a co więcej nie jest usuwany z tej pamięci przez wszystkie iteracje pętli wewnętrznej (może się to odbywać albo ze względu na odpowiednio małą wartość *N*, albo ze względu na wyrafinowany schemat podmiany linii dla większych wartości *N* – w pamięci podręcznej podmieniane są wtedy tylko elementy tablicy `a`, podczas gdy elementy `x` w niej pozostają). W takim przypadku czas dostępów do wektora `x` jest czasem jednokrotnego odczytu wartości wektora z pamięci DRAM do pamięci podręcznej najbliższej rdzeniowi realizującemu obliczenia oraz wielokrotnych odczytów z tej pamięci. Zgodnie z założeniami, czas taki jest znacząco krótszy w porównaniu z czasem odczytu elementów tablicy `a`, zawsze realizowanych z pamięci DRAM, i zazwyczaj może być pominięty w analizie wydajności.

Inaczej sprawa ma się dla większych wartości *N*. W trakcie wykonania zaczynają się pojawiać usunięcia elementów wektora `x` z pamięci *cache* i podmiany linii dla przywrócenia tych wartości w kolejnych iteracjach pętli zewnętrznej. W przypadku odpowiednio dużych wartości *N*, każdy element wektora pobrany w danej iteracji pętli zewnętrznej jest usuwany w tej iteracji z każdej pamięci podręcznej, aby zrobić miejsce dla innych elementów `x` (jest to klasyczny przypadek chybienia pojemnościowego, *capacity miss*, kiedy pamięć podręczna jest zbyt mała, żeby pomieścić cały wektor `x` podczas wszystkich iteracji pętli wewnętrznej). Oznacza to konieczność pobrania wektora `x` (*N* elementów) z pamięci DRAM, w każdej z *N* iteracji pętli zewnętrznej, a więc ostatecznie N^2 odczytów, tak jak w naiwnej analizie opartej na kodzie źródłowym.

Najciekawsze, z punktu widzenia analizy wydajności, są przypadki pośrednie, kiedy wektor `x` mieści się częściowo w pamięciach podręcznych różnych poziomów. Ze względu na złożoność strategii podmiany linii w pamięci *cache*, stosowanej przez różne mikroprocesory, ostateczny przebieg obliczeń i

liczba dostępów do pamięci różnych poziomów, często są w takich przypadkach możliwe do określenia tylko na podstawie danych eksperymentalnych, uzyskanych podczas wykonania kodu.

Analiza kodów asemblera

Badanie czasów wykonania programu każdorazowo warto wzbogacić o analizę realizowanego kodu binarnego, co umożliwia wyciąganie dodatkowych wniosków i pozwala ukierunkować ewentualną dalszą pracę nad optymalizacją na poziomie kodu źródłowego i opcji kompilacji. W przypadku rozważanego algorytmu (a także innych badanych w książce) analiza dotyczy wyłącznie postaci asemblera uzyskanej po optymalizacji automatycznej. Kod bez optymalizacji jest najczęściej trudno czytelny i zawiera wiele operacji zbędnych, w przypadku kiedy celem jest wyłącznie uzyskanie poprawnego wyniku (a nie np. łatwość debugowania).

Kompilator *gcc* uruchomiony z opcją *-O2* produkuje dla najbardziej wewnętrznej pętli algorytmu mnożenia macierz-wektor (ze zmienną tymczasową dla obliczanej wartości elementu wektora `y`) kod asemblera:

```
.L4:
   movsd (%rdi,%rax,8), %xmm0
   mulsd (%rsi,%rax,8), %xmm0
   addq $1, %rax
   addsd %xmm0, %xmm1
   cmpq %rax, %rcx
jne.L4
```

Analiza asemblera wskazuje na sposób potraktowania zmiennych i operacji przez kompilator:

- rejestr `rdi` zawiera adres początkowego elementu aktualnie przetwarzanego wiersza macierzy
- rejestr `rax` zawiera indeks elementu w wierszu
- pierwszy rozkaz, `movsd`, pobiera element macierzy do rejestru `xmm0` (ostatni argument rozkazu, w ramach adresowania złożonego, oznacza, że zmiana indeksu w rejestrze `rax` o jeden powoduje przeskok w pamięci o 8 bajtów, co związane jest z używaniem zmiennych podwójnej precyzji)
- rejestr `rsi` zawiera adres początkowego elementu wektora `x`
- rozkaz `mulsd` powoduje pomnożenie elementu macierzy z rejestru `xmm0` przez pobrany element wektora `x` i zapisanie wyniku w rejestrze `xmm0` (dodatkowe litery `sd`, akronim od *scalar double*, w rozkazach przesunięcia i mnożenia oznaczają, że operacje dotyczą tylko 64 bitów, mimo użycia rejestru 128-bitowego)
- rozkaz `addsd` dodaje obliczony iloczyn do zawartości rejestru `xmm1`, przechowującego tymczasową wartość odpowiadającą elementowi wektora `y` (ponownie efektywnie wykorzystując tylko 64 bity)
- rozkaz `addq` (litera `q` oznacza rozmiar poczwórnego słowa maszynowego, czyli 64 bity) zwiększa o jeden indeks w tablicy `a` i w wektorze `x`, a rozkaz `cmpq` porównuje wartość indeksu z wymiarem *N* zapisanym w rejestrze `rcx`
- w przypadku nieosiągnięcia końca wiersza macierzy, rozkaz `jne` (*jump if not equal*) przenosi sterowanie na początek bloku podstawowego, zaczynającego się od etykiety `L4` i odpowiadającego pojedynczej iteracji pętli wewnętrznej algorytmu, czyli rozpoczyna wykonywanie kolejnej iteracji pętli

Obserwacją, którą można poczynić jest zastosowanie przez kompilator klasycznej optymalizacji CSE. Kod asemblera odpowiada pętli zapisanej w C w postaci:

```
for(j=0; j<N; j++){
  y_i += a[in+j]*x[j];
}
```

gdzie zmienna *in* jest ustawiona na początek przetwarzanego wiersza.

Dalszym modyfikacjom podczas kompilacji poddany jest kod w przypadku użycia opcji *-O3*. Wyprodukowany asembler ma formę:

```
.L5:
   movupd (%rdi,%rax), %xmm0
   movupd (%rsi,%rax), %xmm4
   addq   $16, %rax
   mulpd  %xmm4, %xmm0
   addsd  %xmm0, %xmm1
   unpckhpd   %xmm0, %xmm0
   addsd  %xmm0, %xmm1
   cmpq   %rax, %r9
jne   .L5
```

Najważniejszą różnicą jest wykorzystywanie wszystkich bitów w rejestrach `xmm`. Symbolizują to litery `upd` (*unaligned packed double*) w nazwach rozkazów. Pierwszy rozkaz, `movupd`, pobiera z pamięci do rejestru `xmm0` 128 bitów, a więc dwie liczby podwójnej precyzji z tablicy `a`. Kolejny rozkaz pobiera dwa elementy wektora `x` do rejestru `xmm4`, po czym rozkaz `mulpd` dokonuje mnożenia dla dwóch spakowanych liczb i zapisuje wynik w rejestrze `xmm0`. W efekcie konieczną dodatkową modyfikacją, w stosunku do kodu niezwektoryzowanego, staje się wykonanie sekwencji operacji zmierzających do uzyskania wyniku w pojedynczej wartości skalarnej. Najpierw, za pomocą rozkazu `addsd` dotyczącego 64 bitów, dodawane są wartości z jednej pozycji rejestru `xmm0` do tej samej pozycji rejestru `xmm1`, następnie wykonywana jest operacja rozpakowania i zmiany kolejności liczb w rejestrze `xmm0`, rozkaz `unpckhpd`, a ostateczny wynik uzyskiwany jest ponownie za pomocą rozkazu `addsd`, dodającego do wcześniej uzyskanej wartości skalarnej w rejestrze `xmm1` wartość z rejestru `xmm0`, po zamianie kolejności wartości skalarnych.

Wykonywanie operacji na pełnych rejestrach 128-bitowych, a więc na dwóch liczbach podwójnej precyzji, oznacza że w pojedynczej iteracji przetwarzane są dwa elementy macierzy i wektora. Kompilator zastosował rozwinięcie pętli o czynnik 2, co przejawia się także zmianą adresowania złożonego, gdzie rejestr `rax` zawiera teraz przesuniecie w bajtach od początku wiersza macierzy oraz początku wektora `x`. Zawartość rejestru jest zwiększana w każdej iteracji o 16 (rozmiar w bajtach dwóch liczb 64-bitowych). Cały kod asemblera, poza przedstawionym wyżej blokiem podstawowym, dodatkowo komplikuje się przez konieczność uwzględnienia w innych blokach podstawowych sytuacji kiedy wymiar *N* nie jest podzielny przez 2.

Jeszcze dalej idące zmiany pojawiają się przy zastosowaniu agresywnej optymalizacji ukierunkowanej na pełne wykorzystanie cech mikroarchitektury procesora za pomocą opcji *-O3 -march=core-avx2*. Tym razem analizowany jest asembler wyprodukowany przez kompilator *icx* (następca kompilatora *icc* w środowiskach programowania firmy Intel). Wybór kompilatora związany jest z faktem, że produkuje kod prostszy i łatwiejszy do analizy niż *icc*, który w tym przypadku stosuje dodatkowo rozwiniecie pętli o czynnik 16, co zmniejsza czytelność prezentacji. Asembler stworzony przez *icx* wygląda następująco:

```
.LBB0_12:
```

```
   vmovupd (%rsi,%r11,8), %ymm1
   vfmadd231pd (%rdi,%r11,8), %ymm1, %ymm0
   addq   $4, %r11
   cmpq   %r8, %r11
jb   .LBB0_12
vextractf128   $1, %ymm0, %xmm1
vaddpd   %xmm1, %xmm0, %xmm0
vshufpd   $1, %xmm0, %xmm0, %xmm1
vaddsd   %xmm1, %xmm0, %xmm0
```

Pętla wewnętrzna poddana jest intensywnej optymalizacji zmierzającej do pełnego wykorzystania możliwości przetwarzania wektorowego przez rdzeń z użyciem rejestrów 256-bitowych. Nie tylko pobieranie z pamięci dotyczy 32 bajtów (256 bitów, cztery liczby 64-bitowe), ale także wykonywanie operacji przeprowadzane jest w sposób maksymalnie wydajny, za pomocą wektorowego rozkazu `fmadd` (`ymm0 = ymm1 * mem + ymm0`, gdzie `mem` oznacza wartość pobraną z pamięci). Związane jest to z zastosowaniem rozwinięcia pętli wewnętrznej o czynnik 4, co odzwierciedlone jest zwiększaniem indeksu elementów tablicy `a` (przechowującej macierz $\boldsymbol{A}$) i wektora `x` o 4 w każdej iteracji. Na każdej z pozycji w rejestrze wektorowym `ymm0` sumowane są iloczyny dla co czwartego wyrazu wiersza macierzy $\boldsymbol{A}$ i wektora `x`.

Agresywna optymalizacja przejawia się także tym, że kod pętli wewnętrznej algorytmu wykracza poza ramy pojedynczego bloku podstawowego. Wszystkie iteracje realizowane są z wykorzystaniem wyłącznie rejestrów wektorowych `ymm0` i `ymm1` . Po zakończeniu pętli wynik znajduje się w czterech liczbach podwójnej precyzji, zawierających sumy cząstkowe, spakowanych do pojedynczego rejestru 256-bitowego. Rozpakowanie i wysumowanie wartości, z rejestru `ymm0`, w ostateczności do 64 bitów rejestru `xmm0`, odbywa się nie w każdej iteracji, tak jak było to w przypadku poprzedniej wektoryzacji zastosowanej przez kompilator *gcc*, ale dopiero po zakończeniu całej pętli, za pomocą sekwencji operacji `vextractf128, vaddpd, vshufpd, vaddsd` (gdzie ponownie następuje rozpakowywanie zawartości rejestru i zamiana kolejności elementów w rejestrach, a sumowanie dotyczy najpierw 128 bitów, a następnie 64 bitów[3]). Wartość skalarna z rejestru `xmm0` jest zapisywana do pamięci w odrębnym bloku podstawowym, wykonywanym po realizacji dodatkowej pętli uwzględniającej przypadek N niepodzielnego przez 4 (zgodnie z zasadami optymalizacji *loop unrolling*), za pomocą rozkazu:

```
vmovsd %xmm0, (%rdx,%r10,8)
```

Czasy wykonania

Dla utworzonych rozmaitych wariantów kodu binarnego, odpowiadających różnym opcjom optymalizacji przy kompilacji badanej postaci kodu źródłowego (z podstawieniem do wektora `y` poza wewnętrzna pętlą), przeprowadzone zostały pomiary czasu wykonania na standardowo używanym w książce pojedynczym rdzeniu mikroprocesora Intel Core i7-4790. Dla kompilatora *gcc* i wykonania bez optymalizacji (opcja *-O0*) czas wykonania mnożenia macierz-wektor dla macierzy o wymiarach 10000x10000 (rozmiar ok. 0,8 GB dla zmiennych podwójnej precyzji) wyniósł 0.230s. Odpowiada to wydajności 0,87 Gflop/s, kilkadziesiąt razy mniejszej od teoretycznej maksymalnej do uzyskania na użytym rdzeniu mikroprocesora (wynoszącej ok. 60 Gflop/s, z kilkuprocentowymi różnicami zależnymi od częstotliwości pracy rdzenia).

Po włączeniu optymalizacji (opcje *-O2* i *-O3*) czas ten ulega skróceniu do 0.076s, a wydajność wzrasta do 2,61 Gflop/s. Na uzyskanie jeszcze lepszych wyników pozwala zastosowanie kompilatora *icc*. Dla opcji *-O3* (niezależnie od ewentualnego użycia opcji *-march=core-avx2*) czas wykonania wyniósł

[3] Analizując kod asemblera należy pamiętać, że rejestr `xmm0` stanowi (dolną) część rejestru `ymm0`.

0.045s, a wydajność osiągnęła 4,44 Gflop/s. To wciąż jeszcze ponad dziesięciokrotnie mniej od teoretycznego maksimum dla rdzenia, powstaje więc pytanie co ogranicza wydajność dla tego kodu. Nie jest to wydajność przetwarzania potoków, która teoretycznie dla badanego kodu powinna być wysoka, na co wskazuje postać asemblera z efektywnymi rozkazami wektorowymi, tak dotyczącymi pobierania z pamięci, jak i wykonywania operacji arytmetycznych.

Przeprowadzona wcześniej wstępna analiza wydajnościowa algorytmu wskazuje, że decydującym czynnikiem dla czasu wykonania kodu jest czas pobierania danych. Analiza wskazuje jednocześnie, że charakter pobierania danych jest jednoznaczny w przypadku tablicy `a`, odczyty zawsze dokonywane są z pamięci DRAM (każdy element `a` jest odczytywany jeden raz), natomiast nie do końca określony pozostaje charakter dostępów do elementów wektora `x`. Zależnie od rozmiaru `x` oraz pojemności pamięci podręcznych i szczegółów ich funkcjonowania (np. strategii podmiany linii), dostępy mogą być realizowane zawsze z pamięci DRAM lub w mniejszym lub większym procencie z pamięci podręcznych, bez użycia pamięci DRAM.

W celu rozstrzygnięcia tych wątpliwości można przeprowadzić badania z wykorzystaniem narzędzi pozwalających na szczegółowe ustalenie sposobu funkcjonowania hierarchii pamięci podczas wykonania kodu. Poniżej opisane są eksperymenty z użyciem dwóch narzędzi – programu *valgrind*, przeprowadzającego symulacje pracy mikroprocesora, oraz biblioteki PAPI umożliwiającej dostęp do liczników sprzętowych (opisywanej już w p. 3.9).

Program *valgrind* jest w istocie całym zestawem narzędzi do debugowania i profilowania kodu. Istotą jego działania jest przeprowadzenie symulacji wykonania dla przesłanego jako argument pliku wykonywalnego, połączone ze zbieraniem danych dotyczących poprawności i innych charakterystyk wykonania (np. w podstawowym wariancie pracy, zbierania informacji dotyczących ewentualnych ”wycieków” pamięci, poprawności alokacji i dealokacji pamięci dynamicznej). Symulacje są realizowane rozkaz po rozkazie, dzięki czemu uzyskuje sie bardzo precyzyjne informacje dotyczące wykonywanych rozkazów oraz, jeśli do pliku wykonywalnego dołączona jest informacja debugowania, konkretnych linijek kodu źródłowego. Zakres pobieranych danych zależy od konkretnego używanego narzędzia, które przeprowadza wstępną instrumentację kodu, przed rozpoczęciem symulacji wykonania.

Omówienie funkcjonowania wszystkich narzędzi wchodzących w skład zestawu znacznie wykracza poza ramy książki, jedynym pokrótce scharakteryzowanym poniżej i używanym w dalszej części książki jest narzędzie *cachegrind*. Narzędzie to, zgodnie z filozofią *valgrind*, przeprowadza symulację kodu binarnego, zbierając na podstawie założonego modelu sprzętu informacje o użyciu hierarchii pamięci. Wykorzystywany model jest tylko przybliżeniem rzeczywistej architektury rdzeni i mikroprocesorów (np. używa tylko dwóch poziomów pamięci podręcznej: najbliższej potokom przetwarzania (L1) oraz najdalszej od potoków (LL - *Last Level*), nie uwzględnia *prefetchingu*, itp.), stąd pewne dane mogą być obarczone błędem. Na pewno jednak przeprowadzenie symulacji za pomocą *valgrind* daje szereg istotnych informacji o ogólnym charakterze wykorzystania hierarchii pamięci w kodzie. Zaletą rezultatów zwracanych przez *valgrind* jest ich powtarzalność oraz, w odniesieniu do wybranych wartości, wysoka precyzja.

Uruchomienie programu w wersji zoptymalizowanej (opcja *-O3 -march=core-avx2*) generuje na używanej w książce maszynie testowej podstawowy wydruk:

```
==24206== Cachegrind, a cache and branch-prediction profiler
==24206== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==24206== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==24206== Command: mat_vec_test.exe
==24206==
--24206-- warning: L3 cache found, using its data for the LL simulation.
==24206==
==24206== I   refs:      235,687,648
==24206== I1  misses:          1,392
==24206== LLi misses:          1,380
```

```
==24206== I1  miss rate:        0.00%
==24206== LLi miss rate:        0.00%
==24206==
==24206== D   refs:      150,103,442  (125,035,597 rd   + 25,067,845 wr)
==24206== D1  misses:     37,529,613  ( 25,016,875 rd   + 12,512,738 wr)
==24206== LLd misses:     25,008,622  ( 12,504,752 rd   + 12,503,870 wr)
==24206== D1  miss rate:        25.0% (       20.0%     +       49.9%  )
==24206== LLd miss rate:        16.7% (       10.0%     +       49.9%  )
==24206==
==24206== LL refs:        37,531,005  ( 25,018,267 rd   + 12,512,738 wr)
==24206== LL misses:      25,010,002  ( 12,506,132 rd   + 12,503,870 wr)
==24206== LL miss rate:          6.5% (        3.5%     +       49.9%  )
```

Kolejne linijki wydruku (po wstępnej informacji o programie *valgrind*, jego wersji itp.) informują o:

- nazwie pliku wykonywalnego (w tym przypadku `mat_vec_test.exe`)
- wykryciu pamieci L3 i jej użyciu jako ostatniego poziomu pamięci podręcznej (LL)
- szeregu danych dotyczących pobieranych i wykonywanych rozkazów:
  - całkowitej liczbie rozkazów (`I refs`)
  - liczbie chybień w pamięciach podręcznych I1 i LL przy pobieraniu rozkazów (`I1 misses` i `LLi misses`), wraz z procentowym udziałem każdej z nich w całkowitej liczbie pobranych rozkazów (`I1 miss rate` i `LLi miss rate`)[4]

Następne 5 linii przedstawia najważniejsze informacje z punktu widzenia przeprowadzanej analizy wydajności. Są to parametry dotyczące odniesień do danych (całkowitej liczby oraz jej rozbicia, podawanego w nawiasie, na sumę liczby odczytów (`rd`) i liczby zapisów (`wr`)). Prezentowane wartości odpowiadają:

- bezwzględnej liczbie odniesień do danych (`D refs`)
- liczbie chybień w pamięciach L1 i LL przy dostępach do danych (`D1 misses` i `LLd misses`)
- procentowemu udziałowi chybień w całkowitej liczbie dostępów (`D1 miss rate` i `LLd miss rate`)

Ostatnie linie przedstawiają sumaryczne wartości dotyczące pamięci podręcznej L3 (LL), obejmujące dostępy i do rozkazów, i do danych, wraz z liczbą chybień i jej procentowym udziałem w całkowitej liczbie dostępów.

Z punktu widzenia analizy badanej implementacji mnożenia macierz-wektor istotne są tylko wartości związane z odczytami danych z pamięci. W funkcji mnożenia powinno być realizowane 10000 zapisów elementów wektora `y` do pamięci DRAM (z odpowiednią liczbą chybień w L1 i LL), co jest wartością pomijalną w porównaniu z liczbą odczytów.

Wyniki uwidocznione na wydruku, z dużą liczbą zapisów, dotyczą całości programu, łącznie z inicjowaniem tablic wartościami początkowymi, poza funkcją mnożenia macierz-wektor[5]. Odczyty decydujące o wydajności mnożenia są realizowane tylko w funkcji mnożenia, tak więc z przedstawionego

[4] Symbol `I1` odnosi się do najbliższej rdzeniom pamięci podręcznej rozkazów, różnej od pamięci podręcznej danych `L1` (patrz p. 3.5)

[5] *cachgrind* udostępnia także dane w rozbiciu na poszczególne funkcje, które mogą być wyświetlane za pomocą dodatkowego narzędzia *cg_annotate*. Przyczyną zamieszczenia standardowego wydruku *cachegrind*, jest poza jego czytelniejszą formą, także łatwość, w przypadku badanego kodu, wyodrębnienia danych związanych wyłącznie z funkcją mnożenia macierz-wektor.

wydruku pomijane są informacje odnoszące się do zapisów danych, a także wyniki dotyczące pobierania rozkazów, które, ze względu na prostotę kodu, na pewno nie wpływa na czas wykonania programu.

Całkowita liczba chybień w pamięci L1 – 25 milionów, odpowiada odczytowi 200 milionów zmiennych podwójnej precyzji (każde chybienie odpowiada przeładowaniu linii z ośmioma liczbami). Oznacza to, że w każdej ze 100 milionów iteracji algorytmu pobierane są dwie wartości – jedna z tablicy `a` i jedna z wektora `x`. Żaden z elementów wektora `x` nie pozostaje w pamięci L1 pomiędzy iteracjami pętli zewnętrznej po wierszach macierzy. Inna jest sytuacja w przypadku pamięci L3. Wynik 12,5 miliona chybień oznacza, że dotyczą one tylko elementów tablicy `a` (100 milionów liczb), podczas kiedy wszystkie elementy wektora `x` pozostają w pamięci L3 podczas wykonania wszystkich iteracji zewnętrznej pętli algorytmu. Jest to wynik spodziewany w sytuacji kiedy rozmiar pamięci L3 wynosi 8 MB, a rozmiar wektora `x` tylko ok. 80 kB.

Identyczne wyniki, jeśli chodzi o odczyty danych z hierarchii pamięci, zwracane są przez *valgrind* dla innych opcji optymalizacji. Wyjątkiem są wyniki dla kompilacji bez optymalizacji, gdzie pojawia się dużo nadmiarowych, z punktu widzenia uzyskania poprawnego wyniku końcowego, operacji dostępu do pamięci.

W celu uzyskania bardziej szczegółowych danych, programy odpowiadające różnym opcjom optymalizacji mogą zostać uruchomione z wykorzystaniem wywołań funkcji biblioteki PAPI, zliczających zdarzenia sprzętowe. Jako interesujące, z punktu widzenia badania wydajności implementacji algorytmu mnożenia macierz-wektor dla rdzenia mikroprocesora Intel Core i7-4790 o architekturze Haswell, wykorzystane są zdarzenia:

- L1D.REPLACEMENT – podmiana linii w pamięci L1,
- L2_LINES_IN.ALL – odpowiadające podmianie linii w pamięci L2,

W przypadku zdarzenia L2_LINES_IN.ALL zakłada się, że odpowiada ono pobraniu z pamięci L3 (niezależnie czy bezpośrednio do pamięci L2 jak dla strategii *exclusive caches*, czy do L1, co jednak generuje podmianę linii także w L2, jak np. w przypadku *inclusive caches* i L2 jako *victim cache*).

Biblioteka PAPI (a także narzędzie *perf*), nie są wykorzystywane do szacowania transferu danych z pamięci DRAM na podstawie zliczania zdarzeń sprzętowych, ze względu na występujące problemy (dla platformy sprzętowej stosowanej w książce), omawiane już w p. 4.6.

W przypadku badanego kodu, zliczanie zdarzeń sprzętowych dla pamięci L1 daje wyniki zgodne z wynikami zwracanymi przez symulator *valgrind/cachegrind* (dla tego samego zadania i kodu binarnego). Uzyskana liczba zdarzeń L1D.REPLACEMENT wynosi 25 051 863 co pokrywa się z duża dokładnością z wartością `D1 misses` z *valgrind*.

Wyniki zwracane przez funkcje biblioteki PAPI pozwalają na odpowiedź na nierozstrzygnięte do tego momentu pytanie skąd pobierane były dane do L1 – z L2 czy z L3. Zliczanie wystąpień zdarzenia L2_LINES_IN.ALL, przyjętego jako odpowiadające pobraniu z pamięci L3, zwraca wartość 12 523 265, co odpowiada pobieraniu tylko elementów `a`. Oznacza to, że pojemność pamięci L2 (256kB), wraz z odpowiednią strategią podmiany linii, pozwala na przechowanie wektora `x`, po jednorazowym pobraniu z DRAM, przez cały czas wykonywania mnożenia macierz-wektor.

Uzyskane wyniki pozwalają na liczbowe wyrażenie rozmiarów transferu danych miedzy potokami przetwarzania i różnymi poziomami hierarchii pamięci w trakcie realizacji całego algorytmu.

Rozmiar transferu z L1 do rejestrów, który dotyczy jednego elementu `a` i jednego elementu `x` w każdej iteracji (w sumie 200 milionów liczb podwójnej precyzji), wynosi ok. 1,6 GB (te dane dostępne są już z poziomu analizy kodu źródłowego i asemblera).

Rozmiar transferu z pamięci L2 do L1, odpowiadający chybieniom w L1, uzyskany z badania za pomocą *valgrind/cachegrind* i liczników sprzętowych wynosi także 1,6 GB – 25 milionów podmian linii oznacza pobranie 25x64=1600 milionów bajtów, co ponownie odpowiada 200 milionom liczb podwójnej

precyzji i sytuacji kiedy co ósmy dostęp do tablicy `a` i wektora `x` generuje chybienie w L1. Jest to wynik typowy dla rozmiarów: danych – 8B i pojedynczej linii *cache* – 64B oraz pełnej lokalności przestrzennej dostępów.

Zliczanie zdarzeń sprzętowych pozwala na oszacowanie transferu z L3 do L2 – na poziomie ok. 0,8 GB. Każde chybienie w L1 przy dostępie do tablicy `a` jest jednocześnie chybieniem w L2, natomiast nie występują już w każdej iteracji pętli wewnętrznej chybienia przy dostępie do wektora `x`. Transfer z L3 do L2 maleje dwukrotnie w stosunku do transferu z L2 do L1.

Dla oszacowania transferu z pamięci DRAM przyjąć trzeba wartości wynikające z analizy kodu i symulacji *valgrind*. W obu przypadkach, 100 milionów pobrań ośmiobajtowych elementów `a` i pomijalnej w analizie liczbie pobrań elementów `x` oraz 12,5 milionów chybień w L3, wynikiem jest rozmiar transferu ponownie równy ok. 0,8 GB.

W tym momencie możliwa staje się próba oszacowania czasu wykonania algorytmu. Jednym z modeli wykorzystywanych do tego celu w książce jest model oparty na kilku założeniach:

- w analizowanych algorytmach o czasie wykonania programu (lub jego badanego fragmentu) decydują czasy wykonywania arytmetycznych operacji zmiennoprzecinkowych oraz czasy pobierania danych z hierarchii pamięci, pozostałe operacje realizowane są w tle,
- przyjmuje się, że celem modelowania jest określenie minimalnego możliwego do osiągnięcia czasu wykonania,
- minimalny czas wykonania oblicza się jako najkrótszy z czasów realizacji poszczególnych uwzględnianych operacji, co oznacza dopuszczenie możliwości w pełni niezależnego, równoległego wykonywania operacji,
- minimalny czas dla każdej z operacji oblicza się dzieląc liczbę operacji lub rozmiar transferowanych danych przez maksymalną wydajność dla danej operacji uzyskaną w odpowiednich eksperymentach.

Przyjmując powyższe założenia, minimalny czasu wykonania poszczególnych operacji w kodzie jest szacowany w następujący sposób:

- algorytm wymaga wykonania 200 milionów operacji zmiennoprzecinkowych, maksymalna wydajność, którą można przyjąć jako odpowiadającą warunkom realizacji algorytmu, wynosi, zgodnie z badaniami w p. 3.10, 63 Gflop/s (dzięki wykorzystaniu widocznych w asemblerze wektorowych operacji *fma* i przyjęciu pracy z częstotliwością ok. 4GHz). Obliczony czas jest bliski 0,003s
- czas pobrań z pamięci L1 wynika z ustalonego rozmiaru transferu z L1 do rejestrów – 1.6 GB oraz założonej szybkości transferu. Ponownie można przyjąć optymalną przepustowość dla rozkazów wektorowych i pełnej lokalności przestrzennej uzyskaną w p. 4.5.3, równą ok. 221 GB/s. Daje to w efekcie ok. 0,007s,
- obliczenia czasu pobrań z pamięci L2 korzystają z ustalonego rozmiaru 1.6 GB oraz szybkości transferu 91,3 GB/s (tabela 4.1). Obliczony czas wynosi ok. 0,017s
- z przeprowadzonej analizy wynika, że w trakcie wykonania każde pobranie z L3 jest jednocześnie pobraniem z pamięci DRAM, które wymaga znacznie dłuższego czasu, w związku z czym czas pobrań z L3 można pominąć
- czas pobrań z pamięci DRAM wynosi co najmniej 0,043s, zakładając rozmiar transferu 0,8GB i szybkość 18,6 GB/s (tabela 4.1).

W ostateczności, z przeprowadzonej analizy wynika, że czas wykonania kodu dla badanego przypadku testowego, nie może być krótszy niż 0,043s, co odpowiada wydajności 4,65 Gflop/s. Porównując ten czas z czasem uzyskanym dla najszybszej wersji kodu (kompilator *icc* i opcja kompilacji *-O3*) równym 0,045s, widać, że osiągnięty eksperymentalnie czas jest bliski optymalnemu, a wydajność eksperymentalna 4,44 Gflop/s wynosi ok. 95% optymalnej.

Powstaje pytanie czy w świetle takich wyników warto przeprowadzać dalsze możliwe optymalizacje kodu. Z praktycznego punktu widzenia, uwzględniającego także czynniki ekonomiczne, odpowiedź powinna brzmieć: nie. Istnieje jednak prosta optymalizacja, której zastosowanie może przynieść drobne zyski wydajnościowe, a samo przeprowadzenie jej dla algorytmu mnożenia macierz-wektor może stanowić interesujące ćwiczenie. Ćwiczenie to może okazać się przydatne w przypadku prób zastosowania tej optymalizacji do innych architektur mikroprocesorów, dla których wyniki wydajnościowe wykonania kodu mnożenia macierz-wektor bez tej optymalizacji znacząco odbiegają od optymalnych[6] lub zastosowania dla innych algorytmów (gdzie może przynieść bardziej znaczące korzyści).

Optymalizacja ta, nazywana blokowaniem ze względu na rejestry (*register blocking*), dotyczy w przypadku badanego kodu liczby chybień w pamięci L1. Odpowiadają one za pobrania z pamięci L2, których minimalny czas realizacji w zadaniu testowym wynosi 0,017s (co dotyczy i tablicy `a`, i wektora `x`, a także uwzględnia te chybienia w L2, których czas mieści się w czasie pobrań z pamięci DRAM). Czas ten stanowi tylko ok. 4% całego czasu wykonania algorytmu mnożenia (dla użytego mikroprocesora Intel Core i7-4790). Można więc w celach poglądowych zastosować do kodu źródłowego optymalizację *register blocking*, nie licząc jednak w tym konkretnym przypadku na istotne zyski wydajnościowe.

Optymalizacja *register blocking*

Register blocking (blokowanie pod kątem optymalnego wykorzystania rejestrów) jest techniką stosowaną w przypadku zagnieżdżonych pętli, w których następuje wykorzystanie w każdej iteracji pętli zewnętrznej tych samych elementów jednej lub wielu tablic. Kolejne elementy tablic są odwiedzane w pętli wewnętrznej co powoduje, że domyślnie w każdej iteracji pętli zewnętrznej są one od nowa pobierane do rejestrów (niezależnie czy odbywa sie to z pamieci DRAM czy z dowolnej pamieci podręcznej). Sytuacja taka miała miejsce w każdej z dotychczasowych wersji asemblera dla algorytmu mnożenia macierz-wektor.

Optymalizacja *register blocking* polega na grupowaniu kilku kolejnych iteracji pętli zewnętrznej, stosując rozwinięcie pętli, w taki sposób, żeby po pobraniu wyrazów tablic do rejestrów zostały one wielokrotnie wykorzystane w kilku kolejnych iteracjach. Wymaga to odpowiednich transformacji pętli wewnętrznej, stąd inna nazwa tej optymalizacji *unroll-and-jam*.

Zastosowanie *register blocking* do algorytmu mnożenia macierz-wektor w najprostszym przypadku wygląda następująco:

```
for(i=0; i<N; i+=2){
  y_i = 0.0; y_i1 = 0.0;
  for(j=0; j<N; j++){
    x_j = x[j];
    y_i += a[i*N+j]*x_j;
    y_i1 += a[(i+1)*N+j]*x_j;
  }
  y[i] = y_i; y[i+1] = y_i1;
}
```

[6] Przykładem takiej architektury jest np. mikroprocesor Intel Core i7-9700KF z rdzeniami o architekturze Coffee Lake, dla którego wyniki eksperymentów przedstawione są w dodatku A.

W przedstawionej implementacji zewnętrzna pętla rozwinięta została o czynnik 2, a zmienna `x_j` w zamierzeniu odpowiada pojedynczemu rejestrowi (dla uproszczenia pominięty został dodatkowy kod z optymalizacji *loop unrolling* dla *N* niepodzielnych przez 2).

Zgodnie z wzorem matematycznym pojedynczy element wektora `x` jest używany w sumowaniu dla każdego elementu wektora `y`. Stosownie do tego, w kodzie implementacji naiwnej pobieranie każdego elementu `x` jest dokonywane w każdej iteracji pętli zewnętrznej.

Po optymalizacji *register blocking* każdy element wektora `x` jest nadal pobierany w każdej iteracji pętli zewnętrznej, ale iteracji tych jest teraz dwukrotnie mniej, ze względu na rozwinięcie pętli. W pojedynczej iteracji pobrany element wektora `x` jest użyty dla dwóch kolejnych elementów `y` (a więc wyrazów w dwóch kolejnych wierszach tablicy `a`).

W efekcie kod z *register blocking* gwarantuje, że liczba pobrań z pamięci elementów wektora `x` nie będzie wynosiła N^2, tak jak w wersji naiwnej, ale co najwyżej $N^2/2$. I dotyczy to każdego poziomu pamięci, a więc nie tylko pamięci DRAM, ale także wszystkich poziomów pamięci podręcznej.

Optymalizacja *register blocking* najlepiej funkcjonuje w przypadku rozwinięcia obu pętli, często wspomagając dodatkowo wektoryzację kodu. W przypadku mnożenia macierz-wektor, odpowiadać to będzie wykonywaniu w pojedynczej iteracji pętli wewnętrznej operacji dla małego 2-wymiarowego bloku elementów macierzy $\boldsymbol{A}$. Widać to w przypadku przykładowego kodu dla bloków o rozmiarze 4x2:

```
for(i=0;i<N;i+=4){
  y_i0 = 0;  y_i1 = 0;  y_i2 = 0;  y_i3 = 0;
  for(j=0;j<N;j+=2){
    x_j0=x[j];  x_j1=x[j+1];
    y_i0 += a[i*N+j]    *x_j0 + a[i*N+(j+1)]    *x_j1;
    y_i1 += a[(i+1)*N+j]*x_j0 + a[(i+1)*N+(j+1)]*x_j1;
    y_i2 += a[(i+2)*N+j]*x_j0 + a[(i+2)*N+(j+1)]*x_j1;
    y_i3 += a[(i+3)*N+j]*x_j0 + a[(i+3)*N+(j+1)]*x_j1;
  }
  y[i]  = y_i0;  y[i+1]=y_i1;  y[i+2]=y_i2;  y[i+3]=y_i3;
}
```

Blok rozpoczynający się wyrazem o indeksach *i,j* (w lewym górnym rogu bloku, wyraz ten ma indeks *i*N+j* w tablicy `a`), obejmuje wyrazy w dwóch kolejnych kolumnach i czterech kolejnych wierszach macierzy. Rozwinięcie pętli po indeksie *j* wspomaga wektoryzację z rejestrami 128-bitowymi (dwa elementy w wierszu macierzy w jednym rejestrze wektorowym, podobnie jak dwa elementy wektora `x`, co umożliwia użycie rozkazów wektorowych), a rozwinięcie pętli po indeksie *i* powoduje czterokrotne wykorzystanie każdej z pobranych wartości wektora `x` (co po wektoryzacji oznacza wielokrotne wykorzystanie użytego rejestru wektorowego). Zastosowana w tym przypadku optymalizacja skutkuje znacząco większą liczbą zmiennych tymczasowych (w ostatecznym kodzie binarnym mających prowadzić do wykorzystania rejestrów) niż w przypadku kodu bez optymalizacji.

W tym momencie aspektem, który należy uwzględnić, jest dostępna dla pojedynczego wątku liczba rejestrów. Należy wystrzegać się stosowania zbyt dużych bloków w optymalizacji *register blocking*, co może prowadzić do użycia zbyt dużej liczby zmiennych tymczasowych i w efekcie do wystąpienia efektu *register pressure*, gdzie zamiast korzystania z rejestrów, kompilator jest zmuszony do wprowadzenia odniesień do pamięci (patrz p. 3.10.1). Istotnym w tym względzie czynnikiem jest wektoryzacja – liczba rejestrów wektorowych jest relatywnie duża (np. 16 256-bitowych rejestrów w specyfikacji AVX2) i pozwalają one na przechowanie znacznie większej liczby zmiennych niż rejestry skalarne.

Kod asemblera dla optymalizacji *register blocking*

Kod asemblera po zastosowaniu *register blocking* nie różni się znacząco od kodu przed optymalizacją. W przypadku najprostszym, bloku 2x1, dla opcji *-O2* i kompilatora *gcc* ma postać:

```
.L4:
  movsd (%rdi,%rax,8), %xmm0
  movsd (%rsi,%rax,8), %xmm3
  mulsd %xmm0, %xmm3
  mulsd (%rcx,%rax,8), %xmm0
  addq $1, %rax
  addsd %xmm3, %xmm2
  addsd %xmm0, %xmm1
  cmpq %rax, %r9
jne .L4
```

Występują te same rozkazy co przed optymalizacją, natomiast wykorzystane jest więcej rejestrów. Rejestry `xmm1` i `xmm2` przechowują sumy częściowe dla dwóch elementów wektora `y`, rejestry `xmm0` i `xmm3` są użyte przy wykonywaniu operacji mnożenia. Raz pobrana do rejestru `xmm0` wartość elementu wektora `x` jest użyta dwukrotnie, w dwóch mnożeniach dla dwóch elementów wektora `y`. Rejestr `rdi` wskazuje na początek wektora `x`, natomiast rejestry `rsi` i `rcx` zawierają adresy początków dwóch wierszy macierzy $\boldsymbol{A}$. Indeks dla obu wierszy macierzy i wektora `x` (zmienna *j* w kodzie źródłowym) jest przechowywany w rejestrze `rax`. Wartość *N* znajduje się tym razem w rejstrze `r9`

Dla bardziej agresywnych optymalizacji kod staje się bardziej skomplikowany. W przypadku zastosowania przez kompilator dodatkowego rozwinięcia pętli wewnętrznej, blok podstawowy odpowiadający pojedynczej iteracji jest rozbudowany i zawiera kilkanaście, kilkadziesiąt linii. Zastosowanie opcji *-O3 -march=core-avx2* powoduje użycie przez kompilator rejestrów 256-bitowych, podobnie jak dla wersji przed optymalizacją. Nie zmienia się idea użycia odpowiedniej sekwencji rozkazów, pobierania z pamięci do rejestrów wektorowych (`mov`), wykonywania operacji arytmetycznych na rejestrach (`mul` i `add` lub `fma`) oraz sekwencji operacji dodających sumy częściowe na różnych pozycjach rejestrów wektorowych do pojedynczej skalarnej wartości, połączone z rozpakowywaniem zawartości rejestrów i zmianą kolejności elementów w rejestrach.

Podobnie jak w przypadku kodu przed optymalizacją *register blocking*, łatwą do zdiagnozowania i oceny jest kwestia sposobu wykonywania operacji arytmetycznych, trudniejszą sprawa pobierania danych z pamięci. Odpowiednie obliczenia na bazie kodu źródłowego (lub asemblera) prowadzą do liczby N^2 pobrań skalarnych elementów tablicy `a` i $N^2/2$ pobrań elementów wektora `x`, jednak kwestia realizacji odczytów – czy dokonywane są one zawsze z pamięci DRAM, czy niektóre z wartości są przechowane w pamięci podręcznej pomiędzy iteracjami pętli zewnętrznej, wymaga jak zwykle badania zachowania programu w trakcie wykonania na konkretnym sprzęcie.

Analiza wykonania kodu z optymalizacją *register blocking*

Podczas wykonania kodu na testowym sprzęcie (Intel Core i7-4790) narzędzia badania wydajności zwracają wyniki informujące o zmianach w sposobie wykorzystania zasobów mikroprocesora, w szczególności hierarchii pamięci, po zastosowaniu optymalizacji *register blocking*. Narzędzie *cachegrind* programu *valgrind* produkuje podstawowe dla analizy dane zawarte w standardowym wydruku w postaci:

```
==24013== D   refs:      118,963,396  (93,895,566 rd   + 25,067,830 wr)
==24013== D1  misses:     31,269,611  (18,761,873 rd   + 12,507,738 wr)
==24013== LLd misses:     25,008,622  (12,504,752 rd   + 12,503,870 wr)
```

Wyniki zliczania zdarzeń przy użyciu biblioteki PAPI prezentują się następująco:

```
PAPI event: MEM_UOPS_RETIRED.ALL_LOADS      37509088
PAPI event: L1D.REPLACEMENT                 18767043
PAPI event: L2_LINES_IN.ALL                 14652752
```

W stosunku do wykonania bez optymalizacji *register blocking* można zauważyć kilka istotnych różnic. Wynik zliczania zdarzenia MEM_UOPS_RETIRED.ALL_LOADS potwierdza, możliwą do uzyskania także z analizy asemblera, liczbę pobrań do rejestrów, ok. 37,5 miliona (co daje dla rozkazów wektorowych AVX2 4x37,5=150 milionów liczb podwójnej precyzji). Liczba ta oznacza redukcję ilości pobranych liczb z 200 milionów bez *register blocking*, dzięki dwukrotnemu wykorzystaniu pobranych do rejestru elementów wektora `x` (pobrania elementów macierzy `a` z oczywistych względów pozostają bez zmian). Ze względu na dużą przepustowość transferu z L1 do rejestrów, sama ta redukcja (1.2 GB danych w miejsce 1,6 GB) mogłaby przynieść ok. 0,002 sekundy zysku czasowego.

Bardziej istotne są transfery z pamięci podręcznych bardziej odległych od potoków przetwarzania. W przypadku transferu z L2 do L1 liczba chybień (”*D1 misses*” z *cachegrind*) i przeładowań linii (zliczanych przez zdarzenie L1D.REPLACEMENT) maleje w stosunku do przypadku bez optymalizacji, z ok. 25 milionów, do ok. 18,75 miliona (12,5 miliona chybień dla `a` i tylko 6,25 miliona chybień dla `x`). Spadek objętości transferu, ponownie z 1,6 GB do 1,2 GB, daje tym razem możliwy zysk ok. 0,004 sekundy.

Jeśli chodzi o liczby chybień dla pamięci L2 wskazywane przez liczniki sprzętowe, to pozostają one w sprzeczności z analizami teoretycznymi. Nic nie powinno się zmienić w funkcjonowaniu pamięci podręcznych w przypadku zastosowania *register blocking* – jeśli wektor `x` w całości mieści się w L2 i L3, to powinien zostać tylko raz pobrany, co powoduje liczbę chybień pomijalnie małą z liczbą chybień związaną z tablicą `a` (ok. 12,5 miliona). Wyniki liczników sprzętowych mogą wynikać z uwzględnienia dodatkowych zdarzeń, nie związanych z transferem danych bezpośrednio wskazywanym w kodzie źródłowym, np. wynikających z uwzględnienia złożonych algorytmów pobierania z wyprzedzeniem lub bardziej zaawansowanych szczegółów funkcjonowania pamięci, jak np. zaawansowane strategie podmiany linii w pamięciach *cache*. Zgodnie z zasadami przyjętymi w książce (p. 3.9) za podstawę analiz przyjmowane są dalej wyniki teoretyczne, a rezultaty zliczania zdarzeń traktowane jako odchylenia wskazujące na specyfikę wykonania w konkretnych warunkach (konkretny sprzęt, kod binarny, warunki uruchomienia).

Uzyskany eksperymentalnie czas wykonania funkcji mnożenia macierz-wektor dla przypadku testowego macierzy 10000x10000 i pojedynczego rdzenia mikroprocesora Intel Core i7-4790 wynosi dla kodu z optymalizacja *register blocking* ok. 0.042s. Oznacza to kilkuprocentowy przyrost wydajności w stosunku do przypadku bez tej optymalizacji (4,76 Gflop/s względem 4,44 Gflop/s) i praktycznie wyrównanie czasu wykonania mnożenia macierz-wektor z czasem samego odczytu tablicy `a` z pamięci DRAM, z maksymalną szybkością uzyskaną eksperymentalnie.

Zaprezentowany przykład użycia techniki *register blocking* dla algorytmu mnożenia macierz-wektor pokazuje zastosowanie ręcznej optymalizacji, która zazwyczaj jest zbyt złożona dla kompilatorów. Wprawdzie uzyskane zyski czasowe nie są duże w tym przypadku, jednak sama idea wpływania na liczbę dostępów do pamięci poprzez modyfikacje kodu źródłowego okazuje się możliwa do zrealizowania. Zyski z jej zastosowania zależą od konkretnego algorytmu i mogą być znacznie większe niż dla mnożenia macierz-wektor, co pokazuje kolejny przykład z dziedziny numerycznej algebry liniowej.

5.3.2 Mnożenie macierz-macierz

Mnożenie macierz-macierz (mnożenie macierzy, *matrix multiplication*) jest jednym z najintensywniej studiowanych algorytmów, optymalizowanych dla praktycznie wszystkich architektur systemów

komputerowych. Wynika to częściowo z popularności operacji mnożenia macierzy, wykorzystywanej w wielu bardziej złożonych algorytmach, np. rozwiązujących układy równań liniowych metodami bezpośrednimi, a częściowo ze złożoności analizy wydajności i optymalizacji mnożenia macierzy dla współczesnych mikroprocesorów (w tym procesorów graficznych).

Dla wielu algorytmów algebry liniowej, a także programów z innych dziedzin zastosowań, korzystających wewnętrznie z procedury mnożenia macierz-macierz, optymalizacja wykonania wielowątkowego i ewentualnie wieloprocesowego (dla obliczeń z pamięcią rozproszoną) odbywa się na wyższym poziomie organizacji algorytmu, podczas gdy mnożenie macierzy, często dla bloków stanowiących fragmenty oryginalnych macierzy algorytmu, realizuje pojedynczy wątek. Dlatego analiza i optymalizacja wydajności mnożenia macierzy, także ograniczona tylko do przypadku obliczeń sekwencyjnych, ma kluczowe znaczenie dla wydajności szeregu istotnych algorytmów obliczeniowych.

Definicja problemu i naiwna implementacja algorytmu

Dla ułatwienia, które nie ogranicza ogólności analizy wydajności, rozważane będą macierze kwadratowe o rozmiarze $N \times N$, oznaczane przez $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, z pojedynczymi wyrazami A_{ij}, B_{ij}, C_{ij}, indeksowane od 0 (ze względu na implementację w języku C). Matematyczna definicja mnożenia macierzy jest niezwykle prosta:

$$\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B} \quad \equiv \quad C_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj} \quad i,j = 0,1,...,N-1$$

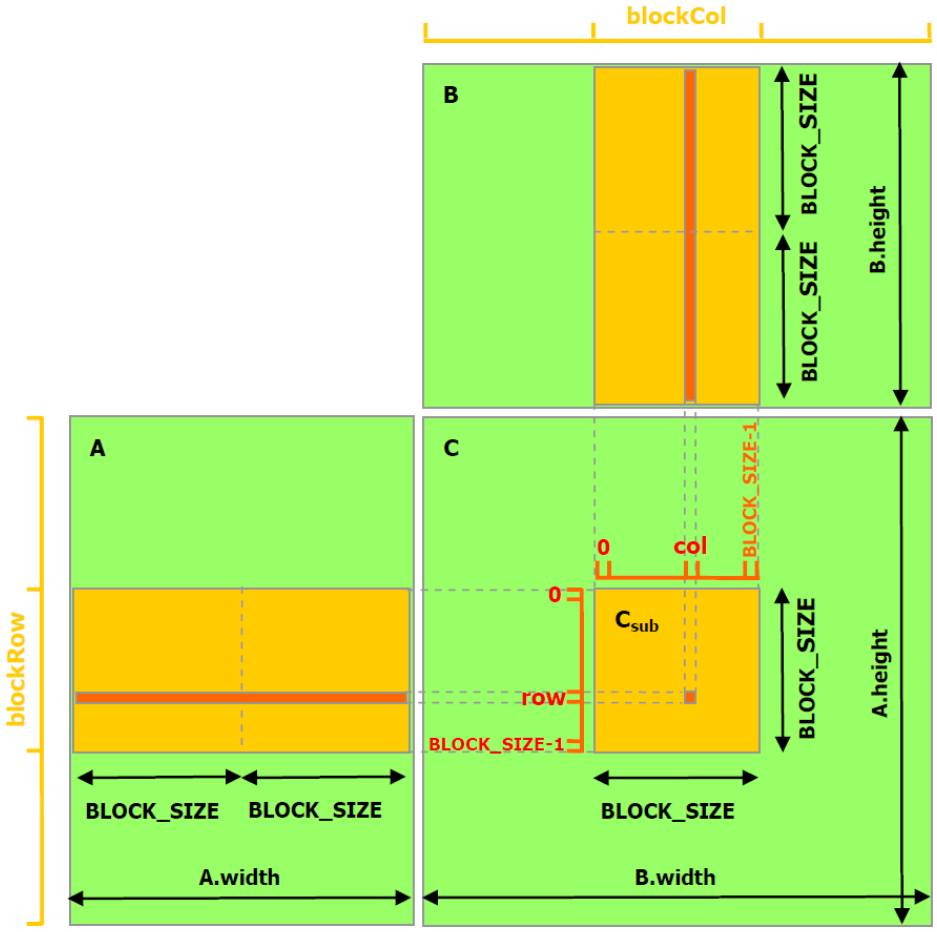
Naiwna implementacja powyższego wzoru, zakładająca przechowywanie macierzy w tablicach dwuwymiarowych ma postać:

```
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    tmp = 0.0;
    for(k=0; k<N; k++){
      tmp += A[i][k] * B[k][j]
    }
    C[i][j] = tmp;
}  }
```

Każdy wyraz tablicy $\boldsymbol{C}$ jest kolejno obliczany jako iloczyn skalarny odpowiedniego wiersza tablicy $\boldsymbol{A}$ i kolumny tablicy $\boldsymbol{B}$. Jedyną modyfikacją w stosunku do wzoru matematycznego jest oczywista optymalizacja (dyskutowana już przy mnożeniu macierz-wektor) przeniesienia podstawienia wartości do macierzy C poza najbardziej wewnętrzną pętlę.

Rysunek 5.1 ilustruje działanie algorytmu, pokazując jak pojedynczy wyraz macierzy $\boldsymbol{C}$ jest obliczany na podstawie odpowiedniego wiersza macierzy $\boldsymbol{A}$ (*row*, w algorytmie indeks *i*) i kolumny macierzy $\boldsymbol{B}$ (*col*, w algorytmie indeks *j*). Pętli po indeksie *k* odpowiadają czerwone paski w macierzach $\boldsymbol{A}$ i $\boldsymbol{B}$. Wymiary macierzy na rysunku uwzględniają możliwość zastosowania operacji mnożenia dla macierzy prostokątnych (przy spełnieniu warunków *B.height=A.width* i wymiarach macierzy wynikowej *A.height* x *B.width*). Rysunek zawiera też szereg innych informacji, które omówione są w dalszej części analizy możliwych implementacji.

Prosta analiza złożoności obliczeniowej algorytmu pokazuje, że wymaga on wykonania $2N^3$ operacji arytmetycznych (N^3 mnożeń i N^3 dodawań) i przechowania $3N^2$ wyrazów macierzy (N^2 wyrazów dla każdej z macierzy). W idealnym przypadku (dyskutowanym już dla mnożenia macierz-wektor) posiadania przez procesor odpowiednio dużej liczby rejestrów, możliwe jest pobranie z pamięci do rejestrów



Rysunek 5.1: Ilustracja mnożenia macierzy w ogólnym przypadku macierzy prostokątnych (objaśnienia w tekście).

obu macierzy wejściowych i wykonywanie wszystkich operacji na rejestrach, z jednokrotnym zapisem każdej wartości macierzy wyjściowej do pamięci. Łącznie daje to $2N^2$ odczytów z pamięci, N^2 zapisów, a więc ostatecznie $3N^2$ dostępów do pamięci. Dla odpowiednio dużych wartości N można oczekiwać, że w takiej sytuacji czas wykonania zależałby od czasu wykonania operacji arytmetycznych, a dostępy do pamięci, mające złożoność rzędu N-krotnie mniejszego, realizowane byłyby w tle.

Liczba rejestrów ogólnego przeznaczenia udostępnianych kompilatorom przez rdzenie współczesnych mikroprocesorów, uwzględniając w tym rejestry wektorowe, jest zbyt mała by przechowywać w nich całe macierze już dla wymiarów N rzędu kilkunastu. Jednak rolę podobną do rejestrów jako szybkiej pamięci lokalnej, pełnią obecnie różne poziomy pamięci podręcznej. W przypadku kiedy zbiór danych na których operuje dany algorytm mieści się w całości w pamięci podręcznej konkretnego poziomu (z najkorzystniejszym przypadkiem pamięci L1, dla której czas dostępu do pojedynczej zmiennej jest zbliżony do czasu wykonania pojedynczej operacji arytmetycznej), można spodziewać się także wysokiej wydajności obliczeń, a znaczenia nabierają optymalizacje dostępu do pamięci podręcznej. Badaniu rozmaitych aspektów takich dostępów (wzorzec dostępu, użycie *prefetchingu*, zastosowanie rozkazów skalarnych lub wektorowych, itp.) poświęcone były rozważania rozdziału 4.

Osobnym problemem jest organizacja algorytmu w przypadku zbiorów danych nie mieszczących się w pamięciach podręcznych. Okazuje się, że przez odpowiednie modyfikacje kodu źródłowego można

uzyskać kontrolę (przynajmniej częściową) nad rozmieszczeniem danych w pamięciach podręcznych i charakterem dostępów do danych podczas wykonania.

W dalszych badaniach algorytmu mnożenia macierz-macierz rozważane są problemy, w których wykorzystywana jest hierarchia pamięci, a więc takie dla których macierze nie mieszczą się w całości co najmniej w pamięci podręcznej poziomu najbliższego rdzeniom (L1). W przykładach w książce algorytmy mnożenia testowane są dla minimalnego wymiaru $N = 2592$ (wybór tej wartości wyjaśniony jest później), co daje ponad 50 MB dla pojedynczej macierzy podwójnej precyzji, a więc dla większości mikroprocesorów więcej niż wynosi pojemność pamięci L3.

Dodatkowo, we wszystkich eksperymentach obliczeniowych używane są (dla wszystkich stosowanych kompilatorów) opcje *-O3 -march=core-avx2*, które maksymalizowały wydajność podczas badania, zbliżonego w formie, algorytmu mnożenia macierz-wektor.

Uruchomienie algorytmu w postaci naiwnej, dla odpowiednio dużej wartości N, powodującej, że macierze nie mieszczą się w żadnej pamięci podręcznej, prowadzi do wydajności wielokrotnie niższej niż teoretyczne maksimum procesora (rdzenia). Dla przykładowego wymiaru $N = 2592$ czas wykonania mnożenia na rdzeniu standardowo używanego w książce mikroprocesora i7-4790 wynosi ok. 59s, co odpowiada wydajności ok. 0.59 Gflop/s. Jest to wartość ponad 100 razy mniejsza od maksymalnej, eksperymentalnie uzyskanej w podrozdziale 3.10 wydajności przetwarzania rdzenia równej 64 Gflop/s (dla częstotliwości pracy ok. 4 GHz – wystepującej w obu przypadkach, testu wydajności przetwarzania rdzenia i mnożenia macierzy).

Taki kod domaga się prób optymalizacji, korzystając z wszystkich dotychczas omówionych technik oraz idących jeszcze dalej. Próby takiej zaawansowanej optymalizacji algorytmu mnożenia macierz-macierz wymagają wszechstronnej analizy szczegółów jego realizacji, która w sekwencji kroków zaprezentowana jest w kolejnych punktach. Analizy początkowo zakładają istnienie tylko jednego poziomu pamięci podręcznej, uwzględniając pełną hierarchię w późniejszych rozważaniach.

Wpływ sposobu przechowywania tablic

Jak było to już analizowane wcześniej, dla zapewnienia ciągłości przechowywania tablic w pamięci operacyjnej, a także w celu jawnego pokazania sposobu dostępu do kolejnych wyrazów tablic w pamięci, niezależnego od szczegółów funkcjonowania konkretnych języków programowania, w algorytmach używane jest przechowywanie macierzy wierszami, w postaci tablic jednowymiarowych. Analiza najbardziej wewnętrznej pętli naiwnego algorytmu mnożenia macierzy w przypadku takiego sposobu przechowywania:

```
for(k=0; k<N; k++){
  tmp += A[i*N + k] * B[k*N + j]
}
```

prowadzi do wniosku, że w kolejnych iteracjach pętli, za każdym razem z tablicy $\boldsymbol{B}$ pobierane są wyrazy odległe o N (taki dostęp z dużym skokiem pomiędzy kolejno odwiedzanymi elementami jest klasycznym przykładem braku lokalności przestrzennej).

Pobranie pojedynczego wyrazu zawsze rozpoczyna się od sprawdzenia odpowiedniego miejsca w linii pamięci podręcznej, albo wyraz razem z całą linią znajduje się już w pamięci podręcznej, albo cała linia musi zostać podmieniona. Brak lokalności przestrzennej algorytmu mnożenia macierzy oznacza, dla odpowiednio dużych wartości *N*, że kolejno odwiedzane wyrazy w pętli po *k* należą do różnych linii *cache*, a dostęp do dowolnego wyrazu następuje w innej iteracji pętli po *j* niż dostęp do któregokolwiek z pozostałych wyrazów w linii. Jest to odzwierciedlenie faktu, że różne wyrazy z linii znajdują się w innych kolumnach macierzy $\boldsymbol{B}$ i dostęp do jednego wyrazu następuje, po przejściu całej długości kolumny (w pętli po indeksie *k*). Inne wyrazy $\boldsymbol{B}$ w pętli po *k* zajmują miejsce w *cache* i w końcu

powodują usunięcie wcześniej pobranego wyrazu wraz z całą linią. W konsekwencji, przy pobieraniu dowolnego wyrazu $\boldsymbol{B}$ następuje chybienie w pamięci podręcznej i konieczność przeładowania linii. Liczba pobranych danych dla macierzy $\boldsymbol{B}$, której pojedynczy element jest odczytywany w każdej z N^3 iteracji algorytmu, powinna zgodnie z kodem wynosić N^3, jednak w rzeczywistości jest równa N^3 pomnożone przez liczbę elementów macierzy w pojedynczej linii pamięci podręcznej (w praktyce najczęściej 8 dla liczb podwójnej precyzji).

Taka sytuacja powinna być uznana za błąd implementacji. Wprawdzie produkowany jest poprawny wynik, jednak wydajność jest znacząco zaniżana przez brak lokalności przestrzennej – w sytuacji kiedy istnieją inne, równie proste jak naiwna, wersje algorytmu, unikające błędu braku lokalności przestrzennej dla macierzy $\boldsymbol{B}$.

Istnienie lokalności przestrzennej odniesień oznacza, że w kolejnych iteracjach najbardziej wewnętrznej pętli odczytywane i zapisywane powinny być wyrazy przechowywane kolejno w pamięci. W celu uzyskania tego efektu można wykorzystać fakt, że w oryginalnej postaci algorytmu mnożenia macierzy, kolejność wykonywania pętli jest dowolna, a więc dopuszczalna jest optymalizacja polegająca na zamianie kolejności pętli (*loop interchange*).

Z postaci algorytmu widać, że dla macierzy przechowywanych wierszami, najbardziej wewnętrzną powinna być pętla po indeksie j:

```
tmp = A[i*N + k];
for(j=0; j<N; j++){
  C[i*N + j] += tmp * B[k*N + j]
}
```

Ta wersja algorytmu zwyczajowo nazywana jest wersją "ikj", w odróżnieniu od naiwnej wersji "ijk". Implementacja "ikj" wymaga wcześniejszego wyzerowania wynikowej tablicy $\boldsymbol{C}$, co jest operacją o czasie zaniedbywalnie małym w stosunku do czasu realizacji mnożenia i w związku z tym pomijaną w dalszych analizach.

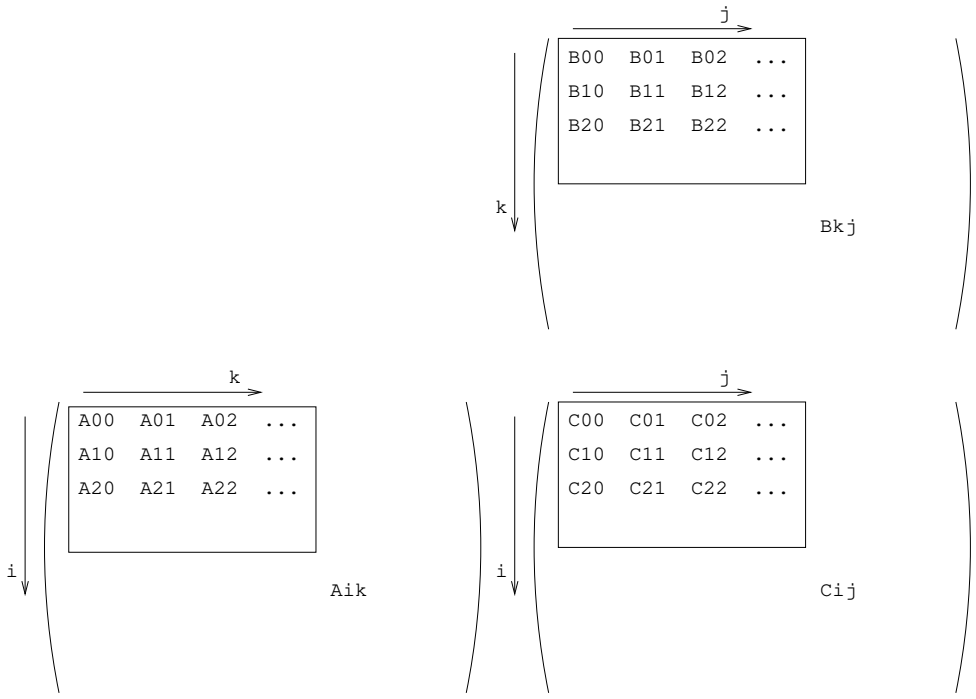
Przeprowadzona optymalizacja powoduje, że w dwóch wewnętrznych pętlach algorytmu (po *j* i po *k*) każdy z wyrazów $\boldsymbol{B}$ jest jednokrotnie pobierany z pamięci głównej do pamięci podręcznej, a następnie wykorzystany w obliczeniach (w odróżnieniu od algorytmu naiwnego, gdzie każdy wyraz był pobierany do *cache* wiele razy, a wykorzystany tylko raz). Implementacja likwiduje brak przestrzennej lokalności odniesień i powoduje kilkukrotne zmniejszenie rozmiaru rzeczywistego transferu elementów tablicy $\boldsymbol{B}$.

Uruchomienie wersji "ikj", dla tych samych parametrów zadania testowego jak dla wersji "ijk", daje czas wykonania 7,99s i wydajność 4,36 Gflop/s , a więc ponad siedmiokrotne skrócenie czasu obliczeń i zwiększenie wydajności.

Wciąż jednak wyniki wydajnościowe są znacznie niższe niż maksymalna wydajność przetwarzania rdzenia. Przyczyną jest fakt, że w najbardziej wewnętrznej pętli ma miejsce $3N$ odniesień do pamięci (dwa odczyty i jeden zapis), co w połączeniu z jednym pobraniem bezpośrednio przed pętlą i N^2-krotnym wykonaniem pętli prowadzi do liczby odniesień $(3N + 1)N^2$.

Gdyby założyć, że wszystkie odniesienia realizowane są z pamięci DRAM (co jak pokażą dalsze analizy zachodzi dla odpowiednio dużych wartości *N*), to występowanie tak dużej liczby odniesień, zbliżonej do liczby wykonywanych operacji arytmetycznych, w połączeniu z tym, że czas pojedynczego odniesienia (nawet w przypadku optymalnym) jest znacznie dłuższy od czasu wykonania pojedynczej operacji arytmetycznej, prowadzi do sytuacji, w której program większość swojego czasu spędza na realizacji dostępów do pamięci.

Przyczyną dla której wszystkie dostępy mogą wymagać korzystania z pamięci DRAM jest fakt, że algorytm wprawdzie wykazuje już lokalność przestrzenną, ale dla odpowiednio dużych wartości N, dla których pojedynczy wiersz tablicy nie mieści się w pamięci podręcznej ma praktycznie nieistniejącą lokalność czasową odniesień. Na przykład dla macierzy $\boldsymbol{B}$, każdy wyraz jest pobrany raz w podwójnej



Rysunek 5.2: Bloki w macierzach $\boldsymbol{A}$, $\boldsymbol{B}$ i $\boldsymbol{C}$ wykorzystywane w optymalizacji *cache blocking*.

pętli po *k* i po *j*. Jednak dla kolejnej iteracji pętli po *i* i każdego wyrazu $\boldsymbol{B}$, mija zbyt dużo czasu, zbyt wiele innych elementów jest pobieranych z pamięci, aby mógł pozostać w pamięci podręcznej. W efekcie w całym algorytmie każdy wyraz tablicy $\boldsymbol{B}$ jest pobierany N razy. Dotyczy to także wyrazów tablicy $\boldsymbol{C}$. Natomiast każdy z wyrazów tablicy $\boldsymbol{A}$ jest pobrany tylko raz, dla odpowiadającej sobie pary indeksów *i* i *k*, a następnie wykorzystany N-krotnie, w całej pętli po indeksie *j*.

Blokowanie ze względu na pamięć podręczna

Prosta analiza przeprowadzona powyżej wskazuje, że sposobem na zwiększenie lokalności czasowej w obliczeniach i zmniejszenie liczby dostępów do tablic w implementacji algorytmu mnożenia macierzy może być technika gwarantująca wielokrotne użycie raz pobranego elementu z pamięci DRAM do pamięci podręcznej (nasuwa się tu podobieństwo z optymalizacją *register blocking*, która jednak dotyczyła wielokrotnego użycia wartości w rejestrach). Ze względu na cel takiej techniki – optymalizację użycia pamięci podręcznej oraz sposób jej realizacji, polegający na grupowaniu wyrazów macierzy w bloki i tworzenie krótkich pętli przebiegających po wyrazach w bloku, technika ta nazywana jest blokowaniem ze względu na pamięć podręczną (*cache blocking*).

W algorytmie "ikj" najbardziej wewnętrzną pętlą jest pętla po indeksie *j* odpowiadająca wierszowi macierzy $\boldsymbol{B}$ i wierszowi macierzy $\boldsymbol{C}$. Pierwszym krokiem analizy może być rozważenie wielokrotnego użycia wyrazów macierzy $\boldsymbol{B}$. Bez utraty ogólności, rozważania można prowadzić dla wyrazu $\boldsymbol{B}_{00}$. Pomocna jest także ilustracja algorytmu na rys. 5.2.

Wyraz $\boldsymbol{B}_{00}$ pobierany jest dla konkretnej wartości indeksu *i*, którą także, podobnie jak indeksy *k* i *j*, przykładowo można przyjąć równą 0. Wyraz $\boldsymbol{B}_{00}$ po pomnożeniu przez $\boldsymbol{A}_{00}$ dodawany jest do wyrazu $\boldsymbol{C}_{00}$, po czym pętla po indeksie *j* przechodzi do sąsiedniego wyrazu $\boldsymbol{B}_{01}$. Dla odpowiednio dużych wartości *N*, kolejno pobierane wartości $\boldsymbol{B}_{0j}, (j = 0, 1, ..., N-1)$ spowodują usunięcie z pamięci podręcznej wyrazu $\boldsymbol{B}_{00}$. Aby temu zapobiec i uzyskać właściwą lokalność czasową algorytmu, pętlę po indeksie *j* należy w pewnym momencie przerwać (np. dla wartości oznaczanej jako *BLS*) i powrócić do

obliczeń korzystających z wyrazu $\boldsymbol{B}_{00}$. Wyraz ten uczestniczy w obliczeniach dla innych indeksów *i*, np. dla indeksu 1, kiedy mnożony jest przez wyraz $\boldsymbol{A}_{10}$, a uzyskany iloczyn dodawany jest do wyrazu $\boldsymbol{C}_{10}$. Oznacza to, że zmiana indeksu *i* musi nastąpić przed kontynuacją pętli po indeksie *j*. Powyższe modyfikacje realizuje następujący kod:

```
for(j=0; j<N; j+=BLS){
  for(k=0; k<N; k++){
    for(i=0; i<N; i++){
      tmp = A[i*N + k];
      for(jj=j; jj<j+BLS; jj++){
        C[i*N + jj] += tmp * B[k*N + jj]
} } } }
```

Oryginalna pętla po indeksie *j* rozbita jest na pętlę wewnątrz pewnego bloku wyrazów w wierszach macierzy $\boldsymbol{C}$ i $\boldsymbol{B}$, z indeksem *jj*, oraz pętlę po indeksie *j*, przebiegającą po początkach kolejnych bloków. Dla ułatwienia algorytm zakłada, że wymiar *N* jest wielokrotnością wymiaru bloku *BLS*. Założenie powyższe jest stosowane w dalszym ciągu analizy dla wszystkich pętli (w innym przypadku należy rozważyć dodatkowe ograniczenia indeksów pętli, tak aby nie przekraczały wartości *N*).

Otrzymany algorytm ma jedną, podstawową wadę. Analizując użycie wyrazu $\boldsymbol{A}_{00}$ (choć dotyczy to każdego wyrazu macierzy $\boldsymbol{A}$), można zaobserwować, że po jego pobraniu i jednorazowym użyciu, zostanie on usunięty z pamięci podręcznej dla pewnej, odpowiednio dużej, wartości indeksu *i* (na skutek zastąpienia przez wyrazy z innych wierszy – indeks *i* przebiega po kolejnych wierszach w kolumnie macierzy $\boldsymbol{A}$). Co więcej, w algorytmie pojawia się brak lokalności przestrzennej dla wyrazów macierzy $\boldsymbol{A}$. Wyrazy pobrane do linii pamięci podręcznej wraz z $\boldsymbol{A}_{00}$ są usuwane z *cache*, zanim zostaną użyte (na skutek usunięcia linii z pamięci podręcznej po to, aby zrobić miejsce dla innych wyrazów z pierwszej kolumny $\boldsymbol{A}$).

Dzieje się tak dla każdego wyrazu macierzy $\boldsymbol{A}$. Kiedy w algorytmie odczytywana jest jego wartość, pobierana jest cała linia pamięci podręcznej. Linia ta jest potem usuwana, zanim jakikolwiek inny wyraz z linii zostanie użyty. Ten brak lokalności przestrzennej wynika z faktu, że pętla po indeksie *k*, jest zewnętrzna względem pętli po indeksie *i*. W celu przywrócenia lokalności przestrzennej, pętla po *k* powinna być wewnętrzna w stosunku do pętli po *i*, tak jak jest to pokazane w kodzie poniżej:

```
for(j=0; j<N; j+=BLS){
  for(i=0; i<N; i++){
    for(k=0; k<N; k++){
      tmp = A[i*N + k];
      for(jj=j; jj<j+BLS; jj++){
        C[i*N + jj] += tmp * B[k*N + jj]
} } } }
```

W efekcie jednak znika wielokrotne wykorzystanie wyrazów macierzy $\boldsymbol{B}$ (np. wyraz $\boldsymbol{B}_{00}$ po użyciu dla pewnego indeksu *i* jest usuwany z pamięci podręcznej przez wyrazy dla kolejnych indeksów *k*). Rozwiązaniem może być umieszczenie wewnątrz każdej iteracji pętli po indeksie *k*, krótkiej pętli po wierszach macierzy $\boldsymbol{A}$ (w kodzie poniżej jest to pętla po indeksie *ii*, o liczbie iteracji *BLS*) oraz pozostawienie na zewnątrz pętli po *k* pętli po początkach bloków, ze względu na indeks *i*:

```
for(j=0; j<N; j+=BLS){
  for(i=0; i<N; i+=BLS){
    for(k=0; k<N; k++){
      for(ii=i; ii<i+BLS; ii++){
```

```
          tmp = A[ii*N + k];
          for(jj=j; jj<j+BLS; jj++){
            C[ii*N + jj] += tmp * B[k*N + jj];
} } } } }
```

W efekcie każdy pobrany wyraz macierzy $\boldsymbol{B}$ (np. $\boldsymbol{B}_{00}$) jest wykorzystany dla sekwencji indeksów *ii*, ale nie jest usuwany z *cache*, gdyż pętla jest krótka i pobieranie wyrazów w kolumnach macierzy $\boldsymbol{A}$ i $\boldsymbol{C}$ nie wymusi usunięcia z pamięci podręcznej wyrazu macierzy $\boldsymbol{B}$. Podobnie każdy pobrany wyraz $\boldsymbol{A}$ (np. $\boldsymbol{A}_{00}$) nie zostanie usunięty z *cache* przez pozostałe wyrazy w kolumnie (bo pętla przebiega tylko przez fragment kolumny) i pobrane razem z nim elementy wiersza macierzy $\boldsymbol{A}$, w momencie kiedy będą używane w algorytmie dla kolejnych indeksów *k*, zostaną pobrane już z pamięci podręcznej.

Ostatecznie, uzyskany kod jest dobrze ilustrowany przez rysunek 5.1. Podwójna wewnętrzna pętla po indeksach *ii* i *jj* oznacza jednokrotną modyfikację wszystkich wyrazów w pewnym dwuwymiarowym bloku elementów macierzy $\boldsymbol{C}$ (na rysunku oznaczonym kolorem żółtym, parametr *BLS* z algorytmu odpowiada oznaczeniu *BLOCK_SIZE* na rysunku). Uczestniczą w tej modyfikacji elementy jednowymiarowego bloku wyrazów w pojedynczej kolumnie macierzy $\boldsymbol{A}$ oraz jednowymiarowego bloku wyrazów w wierszu macierzy $\boldsymbol{B}$. Po aktualizacji bloku macierzy $\boldsymbol{C}$ zmienia się indeks *k* i następuje aktualizacja tego samego bloku, tym razem z udziałem wyrazów z kolejnej kolumny macierzy $\boldsymbol{A}$ i kolejnego wiersza macierzy $\boldsymbol{B}$. Po zakończeniu pętli po indeksie *k*, wszystkie wyrazy w bloku macierzy $\boldsymbol{C}$ są ostatecznie obliczone i algorytm przechodzi do kolejnego bloku macierzy $\boldsymbol{C}$, dla innej pary indeksów *i* i *j*.

Analiza działania algorytmu krok po kroku pozwala na oszacowanie liczby pobrań z pamięci DRAM do pamięci podręcznej dla każdej z macierzy (zakładając dla uproszczenia, że w pamięci podręcznej mieszczą się dwuwymiarowe bloki o rozmiarze *BLS*x*BLS* dla wszystkich trzech macierzy).

Dla macierzy $\boldsymbol{C}$ można rozpocząć od oszacowania dla pojedynczej pary indeksów *i* i *j*, a więc pojedynczego bloku o wymiarach *BLS*x*BLS*. Dla pierwszej iteracji pętli po indeksie *k* ($k = 0$), każdy wyraz bloku jest uaktualniany w podwójnej pętli po indeksach *ii* i *jj*. Zakładając, że pozostaje w pamięci podręcznej w trakcie wykonywania obu pętli, jest on następnie uaktualniany (ponownie używany) dla kolejnych iteracji pętli po *k*. W efekcie każdy wyraz jest ponownie wykorzystywany *N* razy (dla wszystkich iteracji pętli po *k*), a więc ostatecznie jest tylko raz odczytywany z DRAM i raz zapisywany do DRAM. Liczba dostępów do wyrazów całej macierzy $\boldsymbol{C}$ w algorytmie wynosi więc N^2 odczytów i N^2 zapisów.

Analizując pobrania wyrazów tablic $\boldsymbol{A}$ i $\boldsymbol{B}$ ponownie można, bez utraty ogólności rozważań, skupić uwagę na początkowych wyrazach tablic i analizować dostępy dla pojedynczej pary indeksów *i* i *j* (pojedynczego bloku macierzy $\boldsymbol{C}$).

Dla indeksów *k=0* oraz *ii=0* do zmiennej `tmp` pobierany jest element $\boldsymbol{A}_{00}$. Element ten jest wykorzystywany *BLS* razy, we wszystkich iteracjach pętli po indeksie *jj*. Zakładając odpowiednio małą wartość *BLS* (dokładny wymagany rozmiar tych bloków może zależeć od strategii podmiany linii w pamięci podręcznej stosowanej przez rdzeń) element ten pozostaje w pamięci podręcznej dla następnej iteracji pętli po indeksie *k*. Nie jest on w niej użyty, jednak wykorzystany zostaje pobrany wraz z nim sąsiedni element z linii pamięci podręcznej. Tym samym każdy element $\boldsymbol{A}$ jest pobrany raz (albo na skutek dostępu do tego elementu w algorytmie, albo na skutek dostępu do elementu z tej samej linii pamięci podręcznej) i następnie wykorzystany *BLS* razy. Wyraz ten jest pobierany dla każdej iteracji pętli po indeksie *j* (dla każdego nowego bloku kolumn macierzy $\boldsymbol{C}$). Liczba tych iteracji wynosi N/BLS, stąd każdy wyraz $\boldsymbol{A}$ pobierany jest N/BLS razy. W efekcie liczba dostępów (wyłącznie odczyty) do wszystkich wyrazów macierzy $\boldsymbol{A}$ w algorytmie wynosi N^3/BLS.

Analiza dla macierzy $\boldsymbol{B}$ przebiega podobnie. Pojedynczy wyraz (dla pary indeksów *k* i *jj*) jest wykorzystywany *BLS* razy, dla wszystkich iteracji pętli po indeksie *ii* (czyli dla pewnego fragmentu kolumny macierzy $\boldsymbol{A}$ i kolumny macierzy $\boldsymbol{C}$). Wyraz ten jest pobierany dla każdej z N/BLS iteracji pętli po

indeksie i. W konsekwencji ostateczna liczba dostępów (odczytów) dla wszystkich wyrazów macierzy $\boldsymbol{B}$ w całym algorytmie wynosi N^3/BLS.

Jeśli wartości parametru *BLS* sięgają kilkudziesięciu (co wciąż może gwarantować mieszczenie się wszystkich wymaganych bloków nawet w małej pamięci podręcznej L1), oznacza to kilkudziesięciokrotnie mniejszą liczbę dostępów do wolniejszych pamięci bardziej odległych od potoków przetwarzania (pamięci podręcznych wyższych poziomów, a w ostateczności pamięci DRAM), w stosunku do implementacji pozbawionych *cache blocking*. W efekcie czas dostępów do hierarchii pamięci może zostać skrócony kilkudziesięciokrotnie i być zbliżony do czasu wykonywania operacji arytmetycznych, a czas wykonania całego kodu może stać się znacząco krótszy i bliższy optymalnemu.

Ręczna wektoryzacja kodu

Otrzymany kod może być dalej optymalizowany, np. możliwe jest zastosowanie *register blocking* i wektoryzacji. Zastosowanie wektoryzacji można prześledzić dla ostatniej implementacji, a dokładniej dla najbardziej wewnętrznej pętli po indeksie *jj*:

```
tmp = A[ii*N + k];
for(jj=j; jj<j+BLS; jj++){
  C[ii*N + jj] += tmp * B[k*N + jj]
}
```

Możliwe jest zdanie się na wektoryzację automatyczną, realizowaną przez kompilator. Możliwe jest także zastosowanie funkcji wewnętrznych kompilatora (*compiler intrinsics*) i ręczna wektoryzacja. Użycie funkcji wewnętrznych wymaga włączenia do plików kodu źródłowego odpowiednich plików nagłówkowych i użycia odpowiedniego kompilatora. Obie rodziny wykorzystywanych w książce kompilatorów: GNU (*gcc*) i firmy Intel (*icc* i *icx*) wspierają użycie funkcji wewnętrznych.

Pełne omówienie tematyki funkcji wewnętrznych leży poza zakresem tematyki niniejszej książki. Jedynymi omawianymi przykładami są zastosowania w kodzie źródłowym C/C++ funkcji wewnętrznych, odpowiadających pojedynczym rozkazom wektorowym ze zbioru rozkazów AVX2, będącego jednym z rozszerzeń listy rozkazów mikroprocesorów rodziny x86. W tym zastosowaniu funkcje wewnętrzne są prostym sposobem umieszczania w kodzie źródłowym wstawek asemblera służących ręcznej wektoryzacji kodu.

Dla rozważanego fragmentu kodu przykładowa ręczna wektoryzacja ma postać:

```
__m256d v_a = _mm256_broadcast_sd(&A[ii*N+k]);
for(jj=j; jj<j+BLS; jj+=4){
  __m256d v_c = _mm256_load_pd(&C[ii*N+jj]);
  __m256d v_b = _mm256_load_pd(&B[k*N+jj]);
  v_c = _mm256_fmadd_pd(v_a, v_b, v_c);
  _mm256_store_pd(&C[ii*N+jj], v_c);
}
```

Przykład stosuje specjalny typ zmiennych (`__m256d`) odpowiadający 256-bitowym rejestrom wektorowym, wykorzystywanym w kodzie do przechowywania spakowanych zmiennych podwójnej precyzji (cztery zmienne na czterech różnych pozycjach).

Pierwsza użyta funkcja wewnętrzna `_mm256_broadcast_sd` powoduje umieszczenie na wszystkich czterech pozycjach w rejestrze `v_a` pojedynczej zmiennej o indeksie *ii*N+k* z tablicy $\boldsymbol{A}$.

Następnie w pętli po indeksie *jj*, rozwiniętej o czynnik 4, wykonywane są w każdej iteracji operacje dla czterech kolejnych elementów w wierszach tablic $\boldsymbol{C}$ i $\boldsymbol{B}$. Najpierw, za pomocą funkcji `_mm256_load_pd`, cztery kolejne elementy $\boldsymbol{C}$ pakowane są do rejestru `v_c`, a cztery kolejne

elementy $\boldsymbol{B}$ do rejestru `v_b`. Następnie dla rejestrów `v_a`, `v_b` i `v_c` wykonywana jest funkcja `_mm256_fmadd_pd`, z zapisem wyniku ponownie do rejestru `v_c`. Odpowiada to bezpośrednio wykonaniu linijki

```
C[ii*N + jj] += tmp * B[k*N + jj]
```

z kodu przed wektoryzacją, dla czterech kolejnych iteracji, z użyciem operacji *fma*.

Ostatnia linijka w pętli dokonuje zapisu obliczonych zaktualizowanych wartości tablicy $\boldsymbol{C}$ w odpowiednie miejsca w pamięci.

Analiza asemblera

Analiza asemblera jest traktowana w książce jako jeden z najważniejszych etapów badania wydajności. Przeprowadzone w niniejszym rozdziale badanie dla mnożenia macierz-macierz pokazuje korzyści z jej stosowania oraz istotne ograniczenia.

Pierwszą z badanych implementacji jest realizacja naiwna ”ijk”, naśladująca wiernie wzór matematyczny. Kod asemblera uzyskany dla najbardziej wewnętrznej pętli kodu źródłowego, przez kompilację za pomocą *gcc* ze standardowymi używanymi w książce dla algorytmu mnożenia macierz-macierz opcjami *-O3 -march=core-avx2*, ma postać:

```
.L4:
  vmovsd (%rax), %xmm2
  addq $8, %rax
  vfmadd231sd (%rdx), %xmm2, %xmm0
  addq %rcx, %rdx
  cmpq %rax, %rsi
jne .L4
```

Zwraca uwagę kilka faktów związanych z kodem. Nie są używane wektorowe rejestry 256-bitowe, a rejestry 128-bitowe wykorzystywane są tylko dla wartości skalarnych. Świadczy to o niemożności wektoryzacji wykonania. Najprawdopodobniej przyczyną tego jest brak lokalności przestrzennej w dostępach do tablicy $\boldsymbol{B}$ i użycie operacji `fma` z wartością z tablicy $\boldsymbol{B}$ jako argumentem. Sposób obliczenia adresu tej wartości jest drugim istotnym aspektem kodu. Adres przechowywany jest w rejestrze `rdx`, którego zawartość jest zwiększana w każdej iteracji o zawartość rejestru `rcx`. Sugeruje to, że skok pomiędzy elementami $\boldsymbol{B}$ w kolejnych iteracjach pętli nie jest małą, stałą wartością, ale że zależy od parametrów zadania (w tym przypadku wymiaru macierzy N). To zwiastuje istotne wydłużenie czasu wykonania programu.

Podobnych problemów z wykonaniem nie sugeruje postać asemblera (także dla najbardziej wewnętrznej pętli implementacji) uzyskana dla wersji ”ikj”:

```
.L10:
  vmovupd (%rsi,%rax), %ymm0
  vfmadd213pd (%rdx,%rax), %ymm1, %ymm0
  vmovupd %ymm0, (%rdx,%rax)
  addq $32, %rax
  cmpq %r14, %rax
jne .L10
```

Kod jest efektywnie zwektoryzowany, w każdej iteracji pobierane są elementy tablic wypełniające wszystkie 256 bitów rejestrów wektorowych `ymm`, a operacja wykorzystuje optymalny rozkaz wektorowy `fma` (w książce nie są analizowane różnice między rozmaitymi szczegółowymi wariantami operacji `fma`). Skok w adresach danych pobieranych w każdej z iteracji jest jawnie ustalony ns 32 bajty, dodawane do zawartości rejestru `rax`, używanego w adresowaniu pośrednim obu tablic $\boldsymbol{B}$ i $\boldsymbol{C}$.

Ograniczenia analizy z wykorzystaniem kodu asemblera pokazuje jego postać dla implementacji z optymalizacją *cache blocking*:

```
.L5:
  vmovupd (%rsi,%rax), %ymm1
  vfmadd213pd (%rdx,%rax), %ymm0, %ymm1
  vmovupd %ymm1, (%rdx,%rax)
  addq $32, %rax
  cmpq $rcx, %rax
jne .L5
```

Postać ta praktycznie nie różni się od postaci bez *cache blocking* (poza drobnymi zmianami użytych rejestrów). Zaobserwowanie różnicy dla całego algorytmu, między wersją bez optymalizacji i wersją z *cache blocking*, wymaga zbadania kodu asemblera dla pozostałych pętli implementacji. Jest to dodatkowo utrudniane przez zastosowanie przez kompilator, koniecznej w przypadku wektoryzacji, optymalizacji rozwinięcia pętli, która wprowadza dodatkowe bloki asemblera, dla przypadku liczby iteracji niepodzielnej przez czynnik rozwinięcia pętli.

Podobny do powyższego kodu otrzymanego poprzez wektoryzację automatyczną, jest kod asemblera najbardziej wewnętrznej pętli implementacji dla wersji z zastosowaną ręczną wektoryzacją:

```
.L4:
  vmovapd (%rdx), %ymm0
  vfmadd213pd (%rax), %ymm1, %ymm0
  addq $32, %rax
  addq $32, %rdx
  vmovapd %ymm0, -32(%rax)
  cmpq %rax, %rcx
jne .L4
```

Poza użyciem innych wariantów rozkazów wektorowych oraz drobnymi różnicami w obliczaniu adresów, wersje kodu asemblera pozostają w istocie takie same. Sposób użycia pamięci podręcznych nie przejawia się w czytelny sposób w kodzie asemblera. W celu badania ewentualnych różnic w wydajności wykonania związanych z użyciem pamięci podręcznych, konieczne jest użycie innych technik i narzędzi. Zostaną one pokazane po przedstawieniu jeszcze kilku innych możliwości modyfikacji kodu źródłowego.

Dalsze optymalizacje

Wersja kodu z optymalizacją *cache blocking* może być punktem wyjścia dalszych modyfikacji. W tym celu można przedstawić ją w lekko zmienionej postaci (używanej dalej jako podstawowa implementacja), dla której jednak obliczenia przebiegają w sposób prawie identyczny. Modyfikacje polegają na zamianie kolejności pętli po indeksach *i* i *j* oraz rozbiciu pętli po indeksie *k* na dwie pętle (podobnie jak to zostało zrobione dla pętli po indeksach *i* oraz *j*) wraz z zamianą kolejności pętli po *ii* i *kk*:

```
for(i=0; i<N; i+=BLS){
```

```
  for(j=0; j<N; j+=BLS){
    for(k=0; k<N; k+=BLS){

      for(ii=i; ii<i+BLS; ii++){
        for(kk=k; kk<k+BLS; kk++){
          for(jj=j; jj<j+BLS; jj++){

            C[ii*N + jj] += A[ii*N + kk] * B[kk*N + jj]
} } } } } }
```

Sposób zapisu kodu (m.in. rezygnacja ze zmiennej `tmp`) ma na celu pokazanie, że możliwe jest rozbicie potrójnej pętli z oryginalnego kodu (”ijk” lub ”ikj”) na zewnętrzną potrójną pętlę po dwuwymiarowych blokach wszystkich trzech macierzy $\boldsymbol{A}$, $\boldsymbol{B}$ i $\boldsymbol{C}$ oraz wewnętrzną potrójną pętle po wyrazach w blokach. Analiza kodu pokazuje, że dla każdej z potrójnych pętli kolejność pętli składowych jest bez znaczenia dla poprawności wyniku (choć każda kolejność z dużym prawdopodobieństwem prowadzi do innej wydajności obliczeń).

Przedstawiona wersja jest używana w testach wydajności prezentowanych w książce, ze względu na uzyskaną w przeprowadzonych eksperymentach wyższą wydajność w porównaniu do innych możliwych wariantów (z inną kolejnością pętli). Różnice w wydajności nie są duże (sięgają kilkunastu procent) a ich przyczyny tkwią w szczegółach realizacji przez mikroprocesor kodu, który po optymalizacji staje się skomplikowany, powodując złożone wykorzystanie całej hierarchii pamięci podręcznych oraz pamięci podręcznej tablicy stron (TLB).

Sposób wprowadzenia optymalizacji *cache blocking* do oryginalnego kodu wskazuje, że możliwe jest ponowienie całej operacji i dalsze rozbicie wewnętrznej potrójnej pętlę na potrójną pętlę po pewnych blokach (tym razem o rozmiarach mniejszych od *BLS*) w połączeniu z potrójną pętlą po wnętrzach uzyskanych małych bloków.

W taki sposób można przeprowadzać optymalizację *cache blocking* dla pamięci podręcznych różnych poziomów, gdzie bloki mają rozmiary dobrane tak aby duże bloki mieściły się w większej pamięci, dalszej od potoków przetwarzania, a mniejsze bloki w mniejszej pamięci podręcznej, bliższej potokom przetwarzania. Dodawszy do tego fakt, że hierarchia pamięci podręcznych może obejmować trzy poziomy, a wymiary bloków dla pętli po indeksach odpowiadających kierunkom i, j i k mogą być różne, daje to ogromną liczbę, co najmniej kilkuset wariantów optymalizacji. Wybór pomiędzy nimi jest trudny z czysto teoretycznego punktu widzenia (zważywszy na poziom złożoności funkcjonowania pamięci podręcznych, szeroko omawiany w poprzednich rozdziałach książki).

Stosowanym w praktyce rozwiązaniem jest przeprowadzanie, dla każdego wariantu architektury rdzenia i całego mikroprocesora, szeregu eksperymentów dla zestawu różnych wartości parametrów wpływających na wydajność obliczeń, tworzących tzw. przestrzeń parametrów (*parameter space*). Przestrzeń ta jest przestrzenią dyskretną (parametry mają wartości całkowite) i jednym z rozwiązań jest wykonanie eksperymentów dla wszystkich punktów przestrzeni. Nawet ograniczając badania do zbioru wartości parametrów o teoretycznie największym prawdopodobieństwie uzyskania wysokiej wydajności obliczeń, może to prowadzić do bardzo dużej liczby eksperymentów. Z tego względu często stosuje się ograniczenie zbioru punktów w przestrzeni wartości parametrów do pewnego podzbioru, określonego heurystycznie lub np. posługując się technikami sztucznej inteligencji.

Przykładem relatywnie prostego wielopoziomowego blokowania obliczeń jest dodanie do optymalizacji *cache blocking* optymalizacji *register blocking*. Poniżej przedstawiony jest kod, w którym zastosowano *cache blocking*, dla kwadratowych bloków o rozmiarze *BLS*x*BLS*, oraz *register blocking*, dla bloków o rozmiarze 4x4. Rozmiar bloków dla *register blocking* jest dobrany tak, aby umożliwić wektoryzację. Kod przedstawia jawne zastosowanie wektoryzacji za pomocą funkcji wewnętrznych kompilatora (*compiler intrinsics*).

Optymalizacja polega na wielokrotnym użyciu tego samego schematu co przedstawiony wcześniej dla pojedynczej wartości z macierzy $\boldsymbol{A}$ i dwóch czteroelementowych wektorów w wierszach macierzy $\boldsymbol{B}$ i $\boldsymbol{C}$. Tym razem rozważany jest blok 4x4 macierzy $\boldsymbol{A}$ i takie same bloki macierzy $\boldsymbol{B}$ i $\boldsymbol{C}$. Kod przedstawia wektoryzację dla dwóch kolumn w bloku macierzy $\boldsymbol{A}$ oraz dwóch wektorów (wierszy) w bloku macierzy $\boldsymbol{B}$ uczestniczących w modyfikacji całego bloku macierzy $\boldsymbol{C}$. Obliczenia dla pozostałych kolumn bloku $\boldsymbol{A}$ i wierszy bloku $\boldsymbol{B}$ przebiegają w sposób analogiczny:

```
for(i=0;i<N;i+=BLS){
  for(j=0;j<N;j+=BLS){
    for(k=0;k<N;k+=BLS){

      for(ii=i;ii<i+BLS;ii+=4){
        for(kk=k;kk<k+BLS;kk+=4){
         for(jj=j;jj<j+BLS;jj+=4){

            __m256d v_c11 = _mm256_load_pd(&c[ii*N+jj]);
            __m256d v_c21 = _mm256_load_pd(&c[(ii+1)i*N+jj]);
            __m256d v_c31 = _mm256_load_pd(&c[(ii+2)*N+jj]);
            __m256d v_c41 = _mm256_load_pd(&c[(ii+3)*N+jj]);

            __m256d v_a11 = _mm256_broadcast_sd(&a[ii*N+kk]);
            __m256d v_a21 = _mm256_broadcast_sd(&a[(ii+1)i*N+kk]);
            __m256d v_a31 = _mm256_broadcast_sd(&a[(ii+2)*N+kk]);
            __m256d v_a41 = _mm256_broadcast_sd(&a[(ii+3)*N+kk]);

            __m256d v_b11 = _mm256_load_pd(&b[kk*N+jj]);

            v_c11 = _mm256_fmadd_pd(v_a11, v_b11, v_c11);
            v_c21 = _mm256_fmadd_pd(v_a21, v_b11, v_c21);
            v_c31 = _mm256_fmadd_pd(v_a31, v_b11, v_c31);
            v_c41 = _mm256_fmadd_pd(v_a41, v_b11, v_c41);

            __m256d v_a12 = _mm256_broadcast_sd(&a[ii*N+(kk+1)]);
            __m256d v_a22 = _mm256_broadcast_sd(&a[(ii+1)i*N+(kk+1)]);
            __m256d v_a32 = _mm256_broadcast_sd(&a[(ii+2)*N+(kk+1)]);
            __m256d v_a42 = _mm256_broadcast_sd(&a[(ii+3)*N+(kk+1)]);

            __m256d v_b21 = _mm256_load_pd(&b[(kk+1)*N+jj]);

            v_c11 = _mm256_fmadd_pd(v_a12, v_b21, v_c11);
            v_c21 = _mm256_fmadd_pd(v_a22, v_b21, v_c21);
            v_c31 = _mm256_fmadd_pd(v_a32, v_b21, v_c31);
            v_c41 = _mm256_fmadd_pd(v_a42, v_b21, v_c41);

            // podobnie dla kolejnych kolumn w bloku macierzy A
            // i wierszy w bloku macierzy B
            ..........
```

```
            _mm256_store_pd(&c[iin+jj], v_c11);
            _mm256_store_pd(&c[ii1n+jj], v_c21);
            _mm256_store_pd(&c[ii2n+jj], v_c31);
            _mm256_store_pd(&c[ii3n+jj], v_c41);

} } } } } }
```

Eksperymenty obliczeniowe i analiza wykonania

Eksperymenty obliczeniowe przedstawione w niniejszym punkcie dotyczą kilku wybranych implementacji algorytmu mnożenia macierzy z omawianymi dotychczas optymalizacjami. Wszystkie eksperymenty przeprowadzone zostały na standardowo używanej w książce platformie testowej (Intel Core i-7 4790 z kompilatorem gcc[7]) dla macierzy kwadratowych o rozmiarze 2592x2592 (rozmiar ten pozwalał na przeprowadzenie szeregu eksperymentów dla różnych rozmiarów bloków w optymalizacjach *cache blocking* i *register blocking*, przy zachowaniu upraszczającego założenia, że wymiar macierzy jest wielokrotnością wymiaru bloków). Wyniki wydajnościowe eksperymentów przedstawione są w tabeli 5.1.

Kolejne kolumny tabeli zawierają wyniki wydajnościowe, a kolejne wiersze odpowiadają kolejnym wariantom kodu. W pierwszej kolumnie znajduje się opis wariantu implementacji, w drugiej czas wykonania funkcji mnożenia macierz-macierz, a w trzeciej obliczona na podstawie liczby operacji zmiennoprzecinkowych i czasu wykonania wydajność w Gflop/s.

Czwarta i piąta kolumna zawierają liczby zdarzeń chybienia w pamięci podręcznej L1. W czwartej podane są wyniki uzyskane w symulacji *cachegrind/valgrind*, a w piątej wyniki raportowane przez funkcje z biblioteki PAPI dla zdarzenia L1D_REPLACEMENT. Wyniki zwracane przez liczniki sprzętowe, przekraczające (czasem znacząco) liczby podawane przez *cachegrind*, można próbować tłumaczyć, jak to już było wcześniej wskazywane przy analizie implementacji algorytmu mnożenia macierz-wektor, wpływem mechanizmów nieuwzględnianych w modelu *cachegrind*, takich jak np. wyrafinowane strategie podmiany linii lub pobierania z wyprzedzeniem (*prefetching*). Dla dalszej analizy i modelowania, wyniki *cachegrind* traktowane są jako minimalne wartości dla danego zdarzenia, a wyniki PAPI jako wskazówka możliwych odchyleń w trakcie rzeczywistej realizacji obliczeń.

W podobny sposób, jako wskazówka dotycząca rzeczywistego przebiegu obliczeń, rząd wielkości dla analizowanego zdarzenia, a nie jako konkretna wartość, traktowane są przedstawione w kolumnie szóstej wyniki zliczania zdarzenia L2_LINES_IN.ALL, zwracane przez PAPI. Liczby te traktowane są jako odpowiadające chybieniom w pamięci L2 i transferom danych z pamięci L3.

Ostatnia kolumna w tabeli zawiera podawane przez *cachegrind* liczby chybień w pamięci L3, z założenia odpowiadające pobraniom linii pamięci podręcznej z pamięci DRAM. Dla tej operacji, podobnie jak w omawianym wcześniej przypadku mnożenia macierz-wektor, nie są wykorzystywane zdarzenia sprzętowe. Złożoność pobierania z pamięci, rozmaitość związanych z tym zdarzeń sprzętowych, trudności w niezawodnym zliczaniu wykonywanych poza rdzeniem operacji, powodują niemożność znalezienia odpowiednich, wiarygodnych i umożliwiających jednoznaczną interpretację parametrów zwracanych przez liczniki sprzętowe.

Dane z tabeli należy interpretować w świetle charakterystyki zadania testowego. Liczba wszystkich iteracji algorytmu dla przyjętego wymiaru macierzy $N = 2592$ wynosi 17414258688 (w przyjętej w tabeli 5.1 konwencji wyrażania wartości w milionach – 17414), natomiast liczba operacji arytmetycznych jest równa $2 * N^3 \approx 34828$ milionów.

Pierwszą testowaną implementacją jest implementacja naiwna, "ijk". Jak widać z tabeli 5.1 liczba chybień w pamięci podręcznej L1 dla tej implementacji przekracza liczbę iteracji algorytmu. Potwierdza to wcześniejszą analizę, prowadzącą do wniosku, że dla samej tylko tablicy $\boldsymbol{B}$ chybienie następuje w

[7]W dodatku A znajdują się wyniki dla mikroprocesora Intel Core i7-9700KF.

Implementacja	Czas [s]	Wydajność [Gflop/s]	Liczba chybień L1 [10^6]		L. chybień L2 [10^6]	L. chybień L3 [10^6]
			cgrind	PAPI	PAPI	cgrind
naiwna, "ijk"	59,09	0,59	18 024	18 026	17 971	2 178
poprawna, "ikj"	7,99	4,36	4 354	3 944	2 168	2 178
cache blocking - 48	2,38	14,65	149	235	94	47
cache blocking - 48+432	2,02	17,24	113	187	113	11
cache blocking - 48 + 4x12x4	1,22	28,51	190	244	92	47
c. blocking - 48+432 + 4x12x4	0,91	38,16	190	243	96	11

Tablica 5.1: Parametry wykonania na pojedynczym rdzeniu mikroprocesora Intel Core i7-4790 dla kompilatora gcc z opcjami *-O3 -march=core-avx2* oraz różnych implementacji algorytmu mnożenia macierz-macierz.

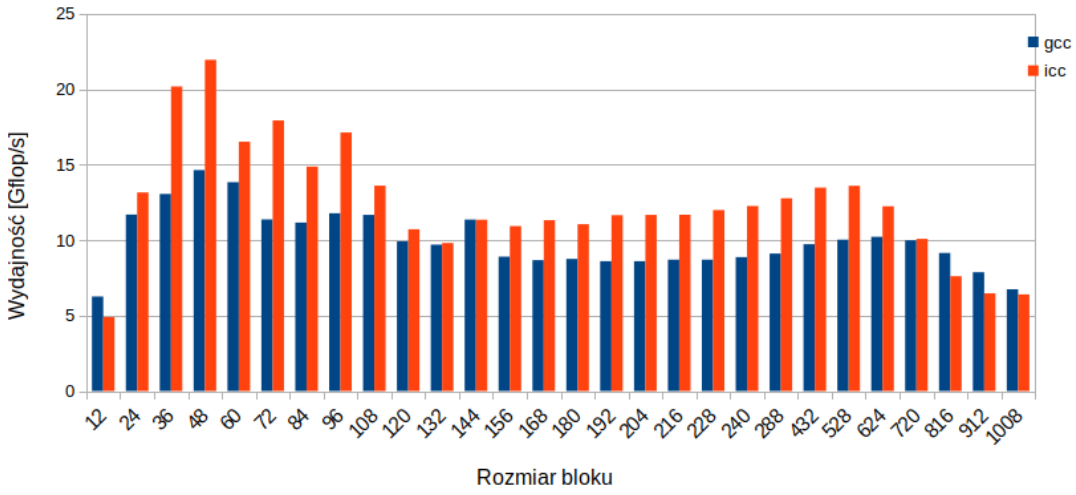
każdej iteracji. Wysoka liczba chybień w L2, praktycznie taka sama jak w L1, wskazuje na nie mieszczenie się odpowiedniej liczby wierszy macierzy w pamięci podręcznej L2. W efekcie każde chybienie w L1 jest chybieniem w L2. Inaczej jest z pamięcią L3, na skutek jej odpowiednio dużego rozmiaru, wystarczającego do pomieszczenia wielu wierszy każdej z macierzy, liczba chybień jest znacznie mniejsza niż dla L1 i L2.

W przypadku implementacji "ikj" sytuacja zmienia się diametralnie dla pamięci podręcznych bliższych potokom przetwarzania. Odzyskanie lokalności przestrzennej dla tablicy $\boldsymbol{B}$, powinno samo w sobie zmniejszyć liczbę chybień w L1 ośmiokrotnie (zamiast chybienia w każdej iteracji jak dla "ijk", dla 8 wartości z $\boldsymbol{B}$ w jednej linii L1 występuje tylko jedno chybienie). Całkowita liczba chybień w L1 obliczona przez *cachegrind* jest prawie dokładnie równa podwojonej liczbie iteracji podzielonej przez 8 (co 8 iteracji dwa chybienia – jedno dla tablicy $\boldsymbol{C}$ i jedno dla $\boldsymbol{B}$). Zbliżony do tej wartości jest wynik zliczania zdarzeń podawany przez funkcje biblioteki PAPI, co świadczy o "naturalnym" przebiegu obliczeń, kiedy rzeczywista praca sprzętu odpowiada modelowi teoretycznemu.

Podobnie jak dla L1, także liczby chybień w L2 i L3 dla algorytmu "ikj", zwracane przez narzędzia badania wydajności, pozwalają na prostą interpretację teoretyczną. W najbardziej wewnętrznej pętli po indeksie *j* odwiedzany jest pojedynczy wiersz tablicy $\boldsymbol{B}$ (co generuje, dzięki lokalności przestrzennej, chybienie w co ósmej iteracji). W kolejnych iteracjach środkowej pętli po indeksie *k* odczytywane są pozostałe wiersze tablicy. Ze względu na fakt, że cała tablica nie mieści się nawet w dużej pamięci L3, na zakończenie pętli po *k* w pamięci podręcznej nie ma już pierwszych wierszy $\boldsymbol{B}$. Kiedy algorytm przechodzi do kolejnej iteracji najbardziej zewnętrznej pętli po indeksie *i* i do obliczeń potrzebne są pierwsze wiersze $\boldsymbol{B}$, muszą być pobrane z pamięci DRAM. Tak więc dla każdej iteracji pętli po *i* (wymagającej odczytu całej macierzy) występuje chybienie w każdym wierszu $\boldsymbol{B}$ (każdej iteracji po *k*) i w co ósmej iteracji po *j* (co ósmym wyrazie w wierszu). Liczba chybień dla tablicy $\boldsymbol{B}$ powinna więc być równa 2592x2592x2592/8≈2176 milionów.

Natomiast w przypadku tablicy $\boldsymbol{C}$ w obu pętlach wewnętrznych odwiedzany jest tylko jej jeden wiersz, który zajmuje 2592*8 bajtów, czyli ok. 20 kB. Taki rozmiar powinien pozwolić na przechowanie go przez cały czas w pamięci L2 i wygenerowanie tylko 2592/8=324 chybień, ponownie dzięki lokalności przestrzennej. Taki mechanizm powtarza się dla każdej iteracji pętli po *i* (dla każdego wiersza tablicy $\boldsymbol{C}$), co prowadzi ostatecznie do liczby chybień równej 324x2592, mniejszej niż milion. Jest to drastyczny spadek w stosunku do liczby chybień (ok. 2176 milionów) z implementacji "ijk".

Podobna liczba chybień jak dla tablicy $\boldsymbol{C}$, powinna wystąpić dla tablicy $\boldsymbol{A}$. Każdy jej wyraz jest wykorzystany wielokrotnie w pętli po *j*, a w pętli po *k* (po wyrazach pojedynczego wiersza macierzy) chybienie następuje co ósmą iterację. Powtarza się tak w każdej iteracji pętli po indeksie *i*, czyli pętli po



Rysunek 5.3: Wydajność implementacji algorytmu mnożenia macierzy z optymalizacją *cache blocking* w zależności od rozmiaru bloków (kompilatory gcc i icc użyte z opcjami *-O3 -march=core-avx2*)

wierszach tablicy $\boldsymbol{A}$. W sumie tak jak dla tablicy $\boldsymbol{C}$ powinno wystąpić 2592x2592/8 chybień.

W ostateczności liczba chybień w tablicach $\boldsymbol{A}$ i $\boldsymbol{C}$ powinna być rzędu 2 milionów, a liczba chybień w tablicy $\boldsymbol{B}$ powinna być równa ok. 2176 milionów jak to zostało wcześniej obliczone. Wyniki zwracane przez symulator *cachegrind* i liczniki sprzętowe potwierdzają słuszność przeprowadzanych analiz.

Na uwagę zasługuje fakt, że obliczona i raportowana przez *cachegrind* liczba chybień w L3, odpowiadająca pobraniom z pamięci DRAM, jest w przypadku algorytmu ”ikj” taka sama jak w przypadku ”ijk”. Wskazuje to na mieszczenie się w pamięci L3 wystarczająco dużej liczby kolumn macierzy $\boldsymbol{B}$, tak że w algorytmie ”ijk” po przejściu całej pętli po indeksie k (po wierszach macierzy $\boldsymbol{B}$ w pojedynczej kolumnie), elementy pobrane na początku pętli do pamięci L3 wciąż się w niej znajdują i dzięki temu są wykorzystane dla kolejnych iteracji pętli po j (cała kolumna odwiedzana w danej iteracji może już znajdować się w cache, dzięki pobraniu w jednej z poprzednich iteracji). W efekcie liczba chybień w L3 spada ok. ośmiokrotnie w stosunku do liczby chybień w L2.

Powstaje pytanie, co powoduje spowolnienie implementacji naiwnej? Istotną rolę, poza samym zwiększeniem liczby chybień w L1 i L2, może odgrywać fakt (możliwy do odczytania z postaci asemblera dla obu implementacji) wykorzystywania dla wariantu ”ijk” operacji skalarnych dostępu do pamięci, podczas gdy wariant ”ikj” używa rozkazów wektorowych.

Kolejny wiersz w tabeli 5.1 dotyczy implementacji z optymalizacją *cache blocking* i rozmiaru bloków 48x48. Rozmiar ten dobrany został w efekcie szeregu eksperymentów z różnymi wielkościami bloków, wyniki których przedstawia wykres 5.3. Wymiary bloków zmieniają się od 12 do 1008 i dla każdego wymiaru przedstawione są dwie kolumny o wysokości odpowiadającej wydajności w Gflop/s, jedna kolumna dla kompilatora *gcc*, a druga dla kompilatora *icc*.

Przedstawione dane pokazują początkowy wzrost wydajności wraz z rosnącym rozmiarem bloków. Teoretyczne analizy, przedstawione w poprzednich punktach, wskazują na zmniejszanie liczby chybień wraz z rosnącym rozmiarem bloków. Powinno to prowadzić do wzrostu wydajności, który jednak na wykresie kończy się dla rozmiaru 60x60. Pojedynczy blok 60x60 zajmuje ok. 28 kB, podczas gdy kolejny badany, 72x72, już ponad 40 kB. Jak widać granicą wzrostu jest mieszczenie się pojedynczego bloku w pamięci L1 (o pojemności 32 kB), co wskazuje na istotne znaczenie dla wydajności właśnie liczby chybień w L1.

Po spadku wydajności po przekroczeniu przez pojedynczy blok pojemności pamięci L1, wydajność

stabilizuje się, aż do wielkości bloków 132x132, a potem ponownie maleje, aż do rozmiaru bloków 168x168, kiedy ponownie ulega stabilizacji, z ewentualną tendencją do lekkiego wzrostu. Ten ostatni rozmiar odpowiada sytuacji kiedy pojedynczy blok (ok. 220 kB) wypełnia prawie całą pamięć podręczną L2 (256 kB). Dla większych rozmiarów bloków wydajność osiąga jeszcze jedno lokalne maksimum dla rozmiaru 528x528, co odpowiada blokom zajmującym ok. 2 MB, a więc sytuacji kiedy trzy bloki mieszczą się w pamięci L3. Po przekroczeniu tego rozmiaru bloków wydajność stopniowo maleje.

Spośród wszystkich analizowanych przypadków kodu z optymalizacją *cache blocking* bloki 48x48 gwarantują najwyższą wydajność. Każdy taki blok zajmuje ok. 18 kB pamięci, co pozwala na przechowanie całego pojedynczego bloku oraz znaczących fragmentów innych bloków w pamięci L1. Eksperymenty pokazują, że wraz z odpowiednią strategią podmiany linii, umożliwia to uzyskanie wysokiej wydajności wykonania.

Przypadek tej wielkości bloków poddany jest dalszym badaniom. Jednak dla implementacji mnożenia macierz-macierz z optymalizacją *cache blocking* analizy nie są już tak szczegółowe i powiązane z rozważaniami teoretycznymi. Implementacja posiada teraz co najmniej sześć pętli, w których podmieniane są elementy wszystkich trzech macierzy. Jak już wspomniane było wcześniej, każda kolejność wykonywania pętli prowadzi do innego wzorca dostępów do tablic $\boldsymbol{A}$, $\boldsymbol{B}$ i $\boldsymbol{C}$ oraz innego schematu podmian linii w pamięciach podręcznych różnych poziomów. Przeprowadzane dalej analizy nie wnikają w szczegóły korzystania z hierarchii pamięci podręcznych, uwzględniają tylko podstawowe fakty oraz ich wpływ na wydajność implementacji.

W ramach takiej podstawowej, uproszczonej analizy można rozważyć przypadek, kiedy bloki o określonym wymiarze *BLS* dla wszystkich trzech macierzy $\boldsymbol{A}$, $\boldsymbol{B}$ i $\boldsymbol{C}$ mieszczą się razem w pamięci podręcznej pewnego poziomu. W ramach algorytmu *cache blocking* potrójna zewnętrzna pętla jest pętlą po blokach. Potrójna wewnętrzna pętla przebiega po wszystkich elementach w blokach.

Przy przejściu do kolejnej iteracji najbardziej wewnętrznej z zewnętrznych pętli (w przedstawianej wcześniej podstawowej implementacji dla optymalizacji *cache blocking* jest to pętla po indeksie *k*) zmieniają się bloki tylko dla dwóch macierzy (we wspomnianym algorytmie są to bloki dla macierzy $\boldsymbol{A}$ i $\boldsymbol{B}$, dla nowej wartości indeksu *k* blok macierzy $\boldsymbol{C}$ pozostaje bez zmian). Oznacza to, że liczba chybień w rozważanej pamięci podręcznej musi być co najmniej równa liczbie iteracji w zewnętrznej potrójnej pętli pomnożona przez podwojoną liczbę chybień koniecznych do przeładowania pojedynczego bloku.

Ta ostatnia wartość jest równa liczbie linii pamięci podręcznej koniecznych do przechowania bloku równa *BLS*x*BLS*/8 (zakładając, że długość pojedynczego wiersza bloku jest wielokrotnością 8, przyjętej jako rozmiar linii w jednostkach liczb podwójnej precyzji). Ostatecznie liczba chybień powinna wynieść co najmniej (N^3/*BLS*^{3})x(2**BLS*x*BLS*/8) = N^3/(*BLS**4).

W przypadku pamięci L1 wymiarem bloków, który jest wielokrotnością 8 i gwarantuje mieszczenie się trzech bloków w L1, jest 32. Dla wymiaru macierzy 2592 obliczenia prowadzą do liczby chybień w L1 równej ok. 136 milionów. Podobne wartości, ponad 138 milionów chybień w L1, daje symulacja wykonania implementacji mnożenia macierzy z takim rozmiarem bloków przeprowadzona za pomocą *cachegrind/valgrind*. Wyniki eksperymentalnego zliczania zdarzenia L1D_REPLACEMENT (240 milionów wystąpień) jak zwykle są wyższe od wartości teoretycznych i otrzymanych z symulacji.

Wyniki wydajnościowe dla implementacji z optymalizacja *cache blocking* i rozmiaru bloków 48x48 prezentuje trzeci wiersz z wynikami w tabeli 5.1. Trzy bloki 48x48 nie mieszczą się w całości w L1, należy więc spodziewać się, że liczba chybień w L1 przekroczy teoretyczną wartość $2592^3/(48*4) \approx$ 90 milionów. Tak też wskazują wyniki z *cachegrind* – 149 milionów, i uzyskane z funkcji PAPI – 235 milionów.

Mimo to, liczba chybień w L1 spada kilkunastokrotnie w stosunku do implementacji bez *cache blocking*, a dodatkowo liczby chybień w L2 i L3 zmniejszają się w jeszcze większym stopniu. Powoduje to ponad trzykrotny wzrost wydajności implementacji.

Kolejnym przypadkiem jest implementacja z dwustopniową optymalizacją *cache blocking*, dla której większe bloki maja rozmiar 432x432 a mniejsze, podobnie jak w poprzednim przypadku, 48x48. Trzy bloki 432x432 mieszczą się w pamięci L3, co daje minimalna liczbę chybień 2592^3/(432*4) $\approx$ 10 milionów. Bliski temu jest wynik symulacji *cachegrind* – ok, 11 milionów. Redukcja liczby chybień w L3 stanowi podstawową zmianę w algorytmie dwustopniowej optymalizacji *cache blocking* w stosunku do poprzedniej optymalizacji jednostopniowej. Inne parametry są zbliżone, a wydajność optymalizacji dwustopniowej rośnie o kolejne kilkanaście procent.

Następne dwie implementacje, dla których parametry wydajności prezentowane są w wierszach 5 i 6 w tabeli 5.1, odpowiadają ręcznie wprowadzonej optymalizacji *register blocking* połączonej z wektoryzacją za pomocą funkcji wewnętrznych kompilatora. Dwa przypadki z wektoryzacją odpowiadają takim samym wariantom optymalizacji *cache blocking*, jak w dwóch przypadkach odpowiadających wierszom 3 i 4 w tabeli wyników.

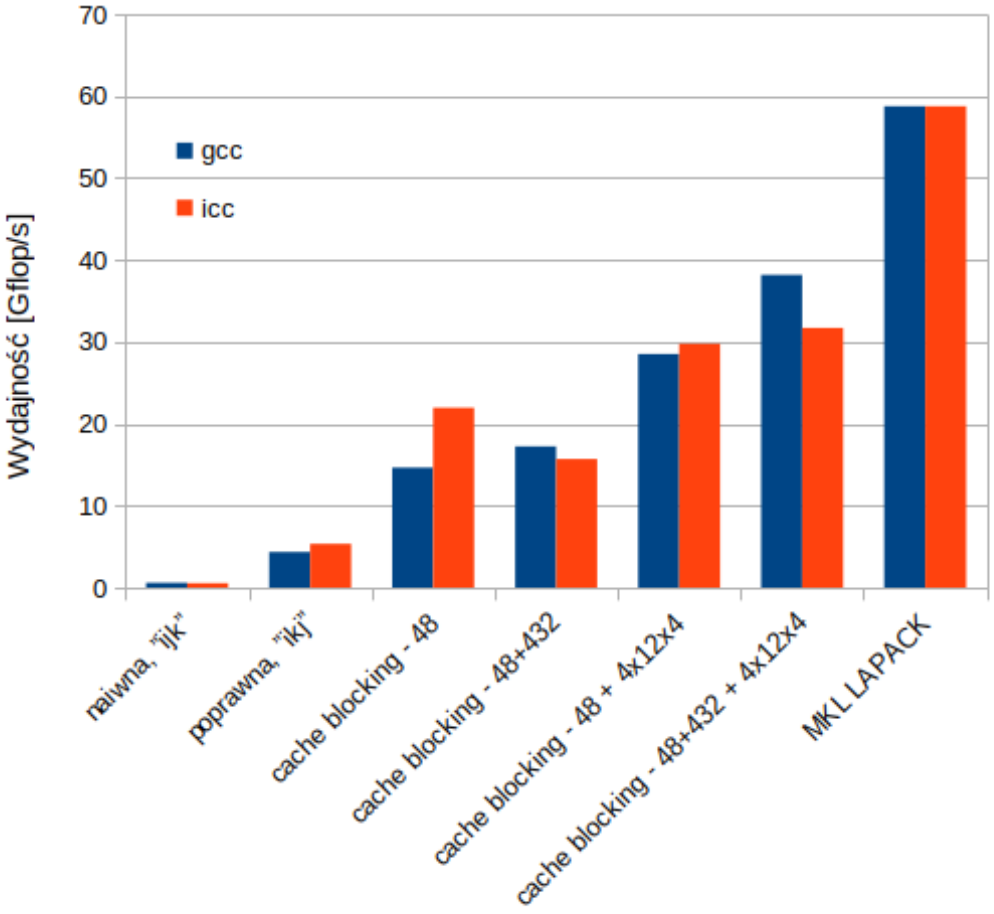
W optymalizacji *register blocking* użyte są bloki 4x12 dla macierzy $\boldsymbol{B}$ i $\boldsymbol{C}$ oraz 4x4 dla macierzy $\boldsymbol{A}$, a idea kodu jest analogiczna jak w zaprezentowanym wcześniej przykładzie wektoryzacji dla bloków 4x4. Większy rozmiar bloków pozwala na użycie większej liczby rejestrów wektorowych i generowanie w jednym bloku kodu, powstałego przez ręczne rozwinięcie potrójnej pętli po wszystkich wyrazach bloków, większej liczby odniesień do pamięci (w intencji ma to spowodować zwiększenie liczby współbieżnie generowanych żądań dostępu do danych). Rozmiar bloków ograniczony jest przez liczbę rejestrów wektorowych – większe bloki prowadzą do zjawiska *register pressure* i konieczności stosowania zwiększonej liczby odniesień do pamięci w kodzie rozwiniętych pętli wewnętrznych.

Liczby chybień w pamięciach podręcznych różnych poziomów pozostają zbliżone do odpowiednich przypadków bez wektoryzacji, jednak wydajność znacząco wzrasta. Pokazuje to jak istotne znaczenie ma staranna organizacja kodu i w konsekwencji przebiegu obliczeń. Kompilator *gcc* stosuje automatyczną wektoryzację z użyciem rejestrów 256-bitowych, jednak to przypadki z ręczną wektoryzacją pozwalają na uzyskanie naprawdę wysokiej wydajności. Ręcznie wektoryzowany kod daje około dwukrotny wzrost wydajności, a także pozwala na uzyskanie racjonalnych proporcji liczby chybień: liczba chybień w L2 jest 2 lub 8-krotnie wyższa od liczby chybień w L3, liczba chybień w L1 jest 2-krotnie (w przypadku danych z liczników sprzętowych 2,5-krotnie) wyższa od liczby chybień w L2.

W ostateczności najwyższa uzyskana wydajność 38,16 Gflop/s (przy częstotliwości pracy rdzenia 4GHz) stanowi ok. 60% teoretycznej maksymalnej wydajności rdzenia dla tej częstotliwości. Zważywszy na stopień złożoności pracy sprzętu podczas wykonania kodu należy ten wynik uznać za zadowalający. Niemniej kwestia optymalizacji algorytmu nie kończy się na *cache blocking*, *register blocking* i wektoryzacji. Staranie zaprojektowane (najczęściej przez producentów sprzętu) funkcje w bibliotekach numerycznej algebry liniowej potrafią osiągać jeszcze wyższe wydajności.

Tak jest w przypadku funkcji z biblioteki MKL (*Math Kernel Library*) firmy Intel, użytej do porównania z przedstawionymi wyżej samodzielnie stworzonymi implementacjami. Biblioteka MKL stanowi realizację standardowo wykorzystywanego w numerycznej algebrze liniowej interfejsu LAPACK. Odpowiadająca iloczynowi macierzy liczb podwójnej precyzji funkcja *dgemm* z biblioteki MKL uzyskuje maksymalną wydajność 58,75 Gflop/s, a więc ponad 90% teoretycznej. Ta wyższa o 50% wydajność w stosunku do zaprezentowanej wcześniej optymalnej własnej implementacji uzyskiwana jest przy zbliżonej liczbie chybień w pamięciach podręcznych, a więc jest wynikiem optymalizacji nie ujętych w analizach prezentowanych w książce (np. optymalizacji użycia pamięci stron).

Niemniej należy wspomnieć, że wyniki wydajnościowe zbliżone do przedstawionego wyżej, udawało się uzyskać dla funkcji z biblioteki MKL tylko w kilku procentach przypadków uruchomienia zadania testowego. Może to wynikać z bardzo dopasowanego do parametrów sprzętu kodu optymalizacji, dla którego drobne zaburzenie w warunkach wykonania programu (np. fakt jednoczesnej realizacji przez system operacyjny innych procesów, korzystających z tych samych zasobów sprzętowych) może prowadzić do zaburzenia zakładanego toku obliczeń. Obserwacja powyższa dotyczy użytego do testów



Rysunek 5.4: Wydajność implementacji algorytmu mnożenia macierzy z kolejno wprowadzanymi optymalizacjami (kompilatory gcc i icc użyte z opcjami *-O3 -march=core-avx2*)

mikroprocesora Intel Core i7-4790, nie będącego mikroprocesorem do zastosowań serwerowych.

Zazwyczaj wydajność funkcji bibliotecznej MKL była niższa od maksymalnej dla najlepszej implementacji własnej i oscylowała w granicach 29-38 Gflop/s. Z kolei wydajność implementacji własnych wykazywała dużą regularność, bez znaczących odchyleń od wartości maksymalnych prezentowanych w tabeli 5.1.

Podsumowaniem analiz wydajności różnych implementacji algorytmu mnożenia macierz-macierz jest diagram 5.4. Porównane są na nim wydajności uzyskiwane przez omawiane powyżej wersje kodu, dla obu podstawowych stosowanych w pracy kompilatorów, *gcc* i *icc* (zawsze używanymi z opcjami *-O3 -march=core-avx2*). Diagram dla porównania przedstawia także maksymalna wydajność, którą w pewnym procencie przypadków udaje się uzyskać dla funkcji z biblioteki Intel MKL.

5.3.3 Wnioski z badania wydajności algorytmów mnożenia macierz-wektor i macierz-macierz

Badanie optymalizacji dla dwóch relatywnie prostych algorytmów numerycznej algebry liniowej, mnożenia macierz-wektor i mnożenia macierz-macierz, pokazuje jak złożone może być to zagadnienie,

nawet w przypadku obliczeń jednowątkowych na pojedynczym rdzeniu współczesnych mikroprocesorów. Niewątpliwe podstawową przyczyną jest tutaj fakt złożoności budowy mikroprocesorów, pojedynczego rdzenia oraz układu pamięci.

Przedstawione przykłady optymalizacji nie wyczerpują wszystkich możliwości, szczególnie w przypadku algorytmu mnożenia macierz-macierz. Pokazuje to porównanie osiągniętych czasów wykonania optymalizowanych wersji implementacji z czasami wykonania dla implementacji w bibliotekach, w szczególności dostarczanych przez producentów sprzętu (w przykładach w książce w bibliotece MKL dla mikroprocesorów firmy Intel).

Sensem podstawowym przedstawienia w niniejszym rozdziale sposobów optymalizacji i analizy wydajności badanych algorytmów pozostaje pokazanie rozmaitych technik i narzędzi, możliwych do zastosowania dla tych i innych algorytmów, w szczególności w przypadku, kiedy niedostępne są zoptymalizowane pod kątem wydajności biblioteki dostarczające wymaganą funkcjonalność, a implementacje w innych bibliotekach nie gwarantują wydajności zbliżonej do optymalnej. Niektóre z tych technik mogą zostać także użyte do badania, czy implementacje, w zewnętrznych bibliotekach lub pochodzące z innych źródeł, spełniają wymagania wystarczającego wykorzystania możliwości wydajnościowych oferowanych przez współczesny sprzęt. Celowi temu służą także metody modelowania wydajności obliczeń opisane w kolejnym rozdziale.

Rozdział 6

Modelowanie czasu wykonania i wydajności programów w obliczeniach sekwencyjnych (jednowątkowych)

Kolejnym rozważanym aspektem tematyki wydajnościowej, odrębnym od teoretycznego i eksperymentalnego badania wydajności wykonywanych programów, jest modelowanie czasu wykonania. Celem takiego modelowania jest uzyskanie wzorów pozwalających na oszacowanie czasu wykonania programu dla przypadków, gdzie nie przeprowadza się eksperymentów obliczeniowych. W stosowanych wzorach pojawiają się parametry zadania oraz kodu źródłowego implementującego badany algorytm, a także teoretyczne lub uzyskane eksperymentalnie parametry sprzętu realizującego obliczenia.

6.1 Czas wykonania programu jednowątkowego

Wszystkie dotychczasowe badania w książce dotyczyły programów jednowątkowych uruchamianych na pojedynczym rdzeniu mikroprocesora. Mimo relatywnej (w stosunku do programów równoległych) prostoty ich wykonania, precyzyjne modelowanie wydajności już w takim przypadku napotyka rozmaite trudności i dokonywane jest najczęściej w sposób przybliżony.

Podstawową przyczyną trudności jest fakt współbieżności pracy rozmaitych elementów sprzętowych. Zakładając możliwość przybliżonego określenia wydajności poszczególnych podukładów realizujących operacje programu, problemem pozostaje uwzględnienie, w jaki sposób wydajności podukładów, decydujące o czasach realizacji konkretnych operacji, wpływają na ostateczny czas wykonania.

Do badania i modelowania czasu wykonania programu można wykorzystać jego kod źródłowy oraz kod asemblera wyprodukowany przez konkretny kompilator z zadanymi opcjami optymalizacji. Kod źródłowy pozwala na obliczenie niezbędnej do uzyskania wyniku liczby wykonywanych operacji, w tym liczby dostępów do zmiennych programu. Nie wystarcza to jednak do ustalenia szczegółów pracy wykonywanej przez wszystkie układy sprzętowe w trakcie wykonania. Szerszy zakres informacji dostarcza analiza asemblera, w szczególności pozwala na ustalenie jakich rozkazów i w jakiej liczbie użył kompilator do realizacji algorytmu zapisanego w kodzie źródłowym.

Analizując kod asemblera wykonywanych programów (lub ich fragmentów), czasem z góry wyróżnić można grupy rozkazów o największym wpływie na wydajność, stanowiące operacje dominujące. Zgodnie z założeniami książki, badane w niej algorytmy charakteryzują się czasami wykonania determinowanymi głównie przez czasy realizacji operacji zmiennoprzecinkowych i dostępów do hierarchii pamięci. W przypadku innych, niż badane w książce, dziedzin zastosowań, wybór operacji, najważniejszych ze względu na wydajność, może dotyczyć innych rozkazów, będących dominującymi w analizo-

wanych algorytmach. Przedstawiana metodologia modelowania wydajności pozostaje w takich przypadkach bez zmian, zmieniają się tylko rodzaje operacji oraz sposoby określania ich liczby i szacowania wydajności przy ich realizacji.

W badanych w książce kodach asemblera pojawiają się, poza rozkazami zmiennoprzecinkowymi i rozkazami dostępów do pamięci, jeszcze rozkazy operacji na liczbach całkowitych, rozkazy porównań, skoków i pewna liczba rozkazów innych typów. Czasy wykonania wszystkich tych operacji są pomijane w modelowaniu wydajności. Wynika to z przyjętego w książce założenia, że wszystkie te rozkazy, są albo wykonywane współbieżnie z operacjami dominującymi (dzięki superskalarnym możliwościom współczesnych mikroprocesorów), a ze względu na mniejszą ich liczbę nie wpływają na czas wykonania, albo, kiedy nie są realizowane współbieżnie z operacjami dominującymi, jest ich na tyle mało, że czas ich realizacji jest pomijalnie krótki.

Oznaczając czas wykonania programu przez T, czas wykonywania operacji arytmetycznych na liczbach zmiennoprzecinkowych przez T_o, a czas realizacji dostępów do pamięci jako T_m, przy powyższych założeniach słusznym staje się oszacowanie postaci:

$$T < T_o + T_m$$

Równość w powyższym wzorze oznaczałaby, że operacje zmiennoprzecinkowe i dostępy do pamięci nigdy nie są wykonywane współbieżnie. Sytuacja taka praktycznie nie jest spotykana.

Znacznie częściej zdarza się, że czynnikiem w pełni określającym czas wykonania programu (lub poddanej badaniu jego części) jest czas wykonywania jednego tylko typu operacji. W dotychczasowych przykładach mikrobenchmarków pojawiały się fragmenty kodu, w których czas wykonania był z dużą dokładnością wyłącznie czasem realizacji przetwarzania potokowego rozkazów zmiennoprzecinkowych, a dostępy do pamięci realizowane były w tle ($T \approx T_o$). Podobnie, dla pewnych innych mikrobenchmarków, występowała sytuacja, kiedy czas wykonania programu determinowany był w całości przez czas operacji na hierarchii poziomów pamięci podręcznej lub na pamięci DRAM, z możliwością pominięcia czasu wykonania pozostałych operacji programu ($T \approx T_m$).

6.1.1 Modelowanie czasu wykonania determinowanego przez wydajność potoków realizacji rozkazów zmiennoprzecinkowych

W celu oszacowania czasu wykonania pewnego fragmentu kodu, dla którego zakłada się, że jest determinowany wyłącznie przez czas wykonywania zmiennoprzecinkowych operacji arytmetycznych (z pobieraniem danych realizowanym w tle), potrzebne jest założenie dotyczące wydajności jaką uzyska sprzęt przy wykonywaniu tych operacji. Wydajność ta jest traktowana jako pewien parametr charakteryzujący sprzęt i określa wydajność uśrednioną dla określonej grupy rozkazów podczas całego wykonywania badanego fragmentu kodu. Jeśli wydajność taką oznaczymy przez W_o, to czas wykonania kodu zawierającego l_o operacji można oszacować jako:

$$T \approx T_o = l_o/W_o$$

We wzorze tym, jeśli wydajność określana jest w standardowej jednostce Gflop/s, uzyskany czas wyrażany będzie w nanosekundach. W dalszych przykładach czas ten zawsze przeliczany jest na sekundy.

Równoważnym podejściem jest użycie odwrotności wydajności w Gflop/s, co prowadzi do definicji wielkości, oznaczanej przez t_o określającej średni czas realizacji pojedynczej operacji zmiennoprzecinkowej:

$$t_o = 1/W_o$$

Ponownie czas ten można bezpośrednio wyrażać w nanosekundach lub przeliczać na sekundy. Czas ten nie odpowiada żadnemu konkretnemu rozkazowi mikroprocesora, ani też grupie rozkazów, odnosi się do

uśrednionego czasu wykonywania matematycznych operacji występujących w algorytmie, najczęściej dodawania i mnożenia liczb.

Koniecznym do uwzględnienia przy szacowaniu czasu wykonania kodu jest fakt, że wydajność W_o silnie zależy od rodzaju rozkazów użytych przez kompilator do realizacji operacji arytmetycznych algorytmu, a także od warunków wykonywania rozkazów. Badania przeprowadzone w p. 4.5 pokazują, że eksperymentalna wydajność pojedynczego rdzenia może różnić się kilkudziesięciokrotnie, w zależności od tego czy między rozkazami (operacjami arytmetycznymi w kodzie) występują zależności oraz od tego czy użyte zostały rozkazy wektorowe. Czasami, z analizy algorytmu i kodu asemblera, można z góry określić, który z przypadków realizacji obliczeń zachodzi, a następnie dobrać stosowny parametr wydajności przetwarzania, np. wydajność uzyskaną w odpowiednio dopasowanym benchmarku.

6.1.2 Modelowanie czasu wykonania determinowanego przez wydajność realizacji operacji na hierarchii pamięci

Do liczbowego wyrażenia czasu wykonania operacji dostępu do hierarchii pamięci używane są w dalszym ciągu rozważań następujące oznaczenia:

- $T_{c1}, T_{c2}, T_{c3}, T_d$ – czas dostępów w trakcie wykonywania programu do danych przechowywanych w pamięciach, odpowiednio: L1, L2, L3, DRAM,
- $W_{c1}, W_{c2}, W_{c3}, W_d$ – wydajność pamięci odpowiednio: L1, L2, L3, DRAM. Wydajność oznacza przyjętą szybkość transferu, wyrażaną w GB/s i uwzględnia złożony charakter dostępu, obejmującego ewentualne podmiany linii w pamięciach bliższych potokom przetwarzania,
- $l_{c1}, l_{c2}, l_{c3}, l_d$ – liczby danych transferowanych odpowiednio z L1, L2, L3, DRAM do potoków przetwarzania. Liczba danych odpowiada omawianej wcześniej sytuacji chybienia w pamięci bliższej potokom przetwarzania i trafienia w danej pamięci (p. 4.6). Liczba danych jest bezwymiarowa i w celu uzyskania objętości transferu należy ją przemnożyć przez parametr $rozmiar\_danej$ wyrażany w bajtach (całość można pomnożyć przez 10^{-9} w celu zyskania objętości w GB).

W przypadkach omawianych wcześniej specjalnych benchmarków, kiedy transfery dotyczyły tylko jednego rodzaju pamięci, np. pamięci podręcznej L2, czas wykonania, równy w tym przypadku czasowi transferu z pamięci L2, mógł być szacowany jako:

$$T \approx T_{c2} = (l_{c2} * rozmiar\_danej * 10^{-9})/W_{c2}$$

Badania przeprowadzone w p. 4.5 pokazują, że nawet w tak prostym modelowym przypadku determinowania czasu wykonania przez dostepy do jednego tylko poziomu w hierarchii pamięci, problemem jest dobór odpowiedniej wydajności (szybkości transferu). Dla przypomnienia, zależnie od sposobu wykonywania operacji (skalarnie, wektorowo) oraz od kontekstu dla konkretnej operacji (dostęp generowany współbieżnie wraz innymi dostępami, dostęp izolowany z powodu braku lokalności przestrzennej i/lub zależności rozkazów dostępu, dostęp możliwy do przewidzenia w ramach mechanizmu *prefetchingu* lub nie), występowały kilkudziesięciokrotne (a w przypadku pamięci DRAM dochodzące nawet do trzystukrotnych) różnice w szybkości transferu. Wybór odpowiedniej wydajności pamięci dla konkretnego algorytmu i konkretnego poziomu w hierarchii pamięci może być trudny, niejednoznaczny i zazwyczaj tylko przybliżony.

W przypadku kiedy podczas wykonania badanego kodu realizowane są dostępy do pamięci różnych poziomów i dla każdego z poziomów dobrana jest właściwa wydajność transferu, czas wszystkich dostępów można ograniczyć przez sumę czasów dla poszczególnych poziomów pamięci:

$$T_m < T_{c1} + T_{c2} + T_{c3} + T_d = 10^{-9} * rozmiar\_danej * (\frac{l_{c1}}{W_{c1}} + \frac{l_{c2}}{W_{c2}} + \frac{l_{c3}}{W_{c3}} + \frac{l_d}{W_d})$$

Równość w powyższym wzorze oznacza brak współbieżności przy dostępach do pamięci różnych poziomów. Trudno wyobrazić sobie taką sytuację, zważywszy na wielość mechanizmów wprowadzających współbieżność do funkcjonowania układu pamięci oraz dążenie sprzętu do maksymalizacji tej współbieżności w trakcie obliczeń.

Próby całościowego modelowania wydajności dostępów do pamięci, zmierzające do uwzględnienia wszystkich poziomów hierarchii pamięci oraz unikania zbytnich uproszczeń, wymagają rozstrzygnięcia szeregu wskazywanych powyżej problemów, wyboru odpowiednich parametrów i wzorów, a następnie właściwej interpretacji obliczonych wyników.

Klasyczna analiza czasu dostępu do pamięci

Klasyczne podejście do szacowania czasu realizacji operacji związanych z pamięcią, posługuje się pojęciem średniego czasu dostępu do pamięci (*average memory access time*, AMAT). Dla sytuacji, kiedy wykorzystywany jest tylko jeden poziom pamięci (czasy dostępów do innych poziomów są pomijalnie małe) można uznać, że taki czas, który np. dla pamięci L1 oznaczyć można jako t_{c1} jest odwrotnością szybkości transferu (po pomnożeniu przez parametry gwarantujące wyrażenie czasu w sekundach):

$$t_{c1} = 10^{-9} * rozmiar\_danej/W_{c1}$$

Jeśli dobrana wydajność odpowiada warunkom realizacji algorytmu można przyjąć, że czas jego wykonania będzie równy:

$$T \approx T_{c1} = l_{c1} * t_{c1}$$

Jeżeli teraz liczbę wszystkich dostępów do danych oznaczymy przez l_m i powyższy wzór pomnożymy oraz podzielimy przez l_m otrzymamy wyrażenie:

$$T \approx T_{c1} = l_m * \left(\frac{l_{c1}}{l_m} * t_{c1} \right)$$

Wyrażenie l_{c1}/l_m, które określa jaki ułamek z całkowitej liczby dostępów zakończył się pobraniem danych z L1 (a więc trafieniem, znalezieniem poszukiwanej wartości), nazywane jest stosunkiem trafień w L1 (*L1 hit ratio*) i w rozważanej sytuacji jest równe 1.

W klasycznym modelu czas t_{c1} określa się jako czas obsługi trafienia w L1, *L1 hit time* (co można oznaczyć jako $t_{c1}^{hit} = t_{c1}$) oraz zakłada się, że brak trafienia oznacza chybienie w L1 i realizację sekwencji operacji obsługi chybienia.

Przyjmując, że chybienie w L1 prowadzi do podmiany linii w L1 i ponownego pobrania z L1, można wyróżnić czas samej podmiany linii jako narzut związany z chybieniem (t_{c1}^{miss}, *miss penalty*). Prowadzi to do wzoru, w którym zakłada się, że liczba chybień jest równa $l_m - l_{c1}$, a czas w przypadku chybienia jest sumą $t_{c1}^{hit} + t_{c1}^{miss}$:

$$T_m = \left(l_{c1} * t_{c1}^{hit} + (l_m - l_{c1}) * (t_{c1}^{hit} + t_{c1}^{miss}) \right)$$

Ponowne pomnożenie i podzielenie każdego ze składników przez l_m oraz wyciągnięcie l_m przed nawias prowadzi do oszacowania:

$$T_m = l_m * \left(\frac{l_{c1}}{l_m} * t_{c1}^{hit} + \frac{(l_m - l_{c1})}{l_m} * (t_{c1}^{hit} + t_{c1}^{miss}) \right) = l_m * \left(t_{c1}^{hit} + \frac{(l_m - l_{c1})}{l_m} * t_{c1}^{miss} \right)$$

Uzyskana po przekształcenia algebraicznych ostateczna postać wzoru wskazuje na fakt, że w ostateczności wszystkie dostępy realizowane są z L1, a narzut dotyczy tylko chybień w L1. Iloraz liczby chybień, równej $l_m - l_{c1}$, przez całkowitą liczbę dostępów l_m, oznaczany dalej przez r_{c1}^{miss}, określić można jako procent chybień w L1 (*L1 miss ratio*):

$$r_{c1}^{miss} = \frac{(l_m - l_{c1})}{l_m}$$

Parametr ten jest często podawany przez różne narzędzia badania wydajności kodu (np. był widoczny na wydrukach wyników programu *valgrind*, dla omawianych wcześniej eksperymentów z mnożeniem macierz-wektor i macierz-macierz).

Jeśli teraz podzielimy obie strony otrzymanego wzoru na czas wykonania T_m przez l_m i wykorzystamy oznaczenie r_{c1}^{miss}, to uzyskamy klasyczny wzór na średni czas dostępu do pamięci:

$$\text{AMAT} = \frac{T_m}{l_m} = t_{c1}^{hit} + r_{c1}^{miss} * t_{c1}^{miss}$$

Uzyskane wyrażenie ma zaletę prostoty i łatwości stosowania w przypadku znajomości występujących w nim parametrów. Relatywnie najprostszym do uzyskania jest procent chybień, otrzymywany dla wykonywanych programów dzięki odpowiednim narzędziom, w tym np. licznikom sprzętowym, a dla kodu źródłowego na podstawie odpowiedniej analizy (jak np. przeprowadzane w książce rozważania dla algorytmów mnożenia macierz-wektor i macierz-macierz).

Trudniejszym do oszacowania jest parametr czasu obsługi trafienia w L1. Próbując określać go należy uwzględniać wszelkie czynniki wpływające na jego wielkość, wymieniane już wcześniej (skalarność lub wektorowość dostępu, lokalność przestrzenną, zależności między rozkazami dostępów, itp.).

Jeszcze trudniejszą jest kwestia szacowania czasu obsługi chybienia. Chybienie w L1 może prowadzić do trafienia w L2, w L3 lub dostępu do pamięci DRAM. W każdym z tych przypadków czas jest inny. W klasycznej analizie rozważa się rekurencyjne stosowanie wzoru ma średni czas dostępu, gdzie czas chybienia w L1 jest czasem trafienia w L2 lub czasem obsługi chybienia w L2 i podobnie dla L3. Prowadzi to do wzoru:

$$\text{AMAT} = t_{c1}^{hit} + r_{c1}^{miss} * \left(t_{c2}^{hit} + r_{c2}^{miss} * \left(t_{c3}^{hit} + r_{c3}^{miss} * t_{c3}^{miss} \right) \right)$$

Wzór daje prosty i czytelny sposób szacowania czasów dostępu do pamięci, jednak próby określania czasów obsługi chybienia, a nawet czasów obsługi trafienia, dla pamięci podręcznych bardziej odległych od potoków przetwarzania napotykają istotne trudności. Zważywszy, że czas ten nie obejmuje czasu dostępu do pamięci podręcznych bliższych potokom przetwarzania, nie jest możliwe bezpośrednie uzyskanie go z odpowiednich benchmarków. Wyniki przedstawione w p. 4.5, dotyczące opóźnienia i przepustowości dla różnych poziomów pamięci, wymagają dalszych analiz i przeliczeń przed zastosowaniem do obliczania średniego czasu dostępu AMAT za pomocą wzoru przedstawionego powyżej.

Nie są to jedyne trudności przy stosowaniu klasycznego modelu szacowania czasu dostępów do pamięci za pomocą czasu AMAT. Główną wadą modelu jest założenie sekwencyjnej realizacji dostępów do różnych poziomów pamięci (w trakcie obsługi chybienia w L1 nie następuje realizacja innych prób dostępu do L1). Jest to sprzeczne ze sposobem funkcjonowania układów pamięci współczesnych mikroprocesorów, które pozwalają na współbieżne realizowanie wielu żądań, z których część może być trafieniami, a część chybieniami.

Ten fakt powoduje, że w dalszej części książki klasyczna analiza nie jest używana. W jej miejsce stosowany jest prostszy model, uwzględniający tylko dostępy do pamięci DRAM, ewentualnie przeprowadzane są bardziej złożone badania wykorzystujące bezpośrednio wyniki benchmarków mierzących wydajność różnych poziomów hierarchii pamięci.

6.2 Model roofline

Wspomnianym w poprzednim punkcie prostszym modelem wydajnościowym jest tzw. model *roofline*. Za punkt wyjścia rozważań dotyczących modelu *roofline* można przyjąć wyprowadzony wcześniej wzór na ograniczenie czasu wykonania programu:

$$T < T_o + T_m$$

Równość w powyższym wzorze, odpowiadająca brakowi współbieżności przy wykonywaniu operacji arytmetycznych i dostępów do pamięci, jest jedną ze skrajności w możliwym modelowaniu. Drugą skrajnością jest przyjęcie pełnej współbieżności, kiedy operacje arytmetyczne wykonywane przez potoki przetwarzania i dostępy do pamięci są realizowane niezależnie, a o czasie wykonania decyduje dłuższy z czasów T_o i T_m:

$$T \geq \max(T_o, T_m)$$

W przypadku modelowania samego czasu dostępów do pamięci T_m, za pomocą czasów poświęconych na dostępy do pamięci różnych poziomów, ponownie można rozważyć przypadek braku współbieżności, co prowadzi do wzoru:

$$T_m = T_{c1} + T_{c2} + T_{c3} + T_d$$

lub przypadek pełnej współbieżności przy dostępie do pamięci różnych poziomów, co daje oszacowanie:

$$T_m \geq \max(T_{c1}, T_{c2}, T_{c3}, T_d)$$

Upraszczającym założeniem w podstawowym modelu *roofline* jest pominięcie czasów dostępu do pamięci podręcznych i używanie wyłącznie czasu dostępu do pamięci DRAM:

$$T_m \geq T_d = 10^{-9} * rozmiar\_danej * \frac{l_d}{W_d} = l_d * t_d$$

gdzie t_d oznacza uśredniony czas dostępu do pojedynczej zmiennej w pamięci DRAM.

W modelu *roofline* podstawowe oszacowania dotyczą nie czasu wykonania programu, ale wydajności przy jego realizacji. Model *roofline* powstał początkowo na potrzeby obliczeń naukowo-technicznych, gdzie spośród wykonywanych operacji dominują operacje na liczbach zmiennoprzecinkowych, a wydajność zwyczajowo wyrażana jest w Gflop/s. Znając liczbę operacji zmiennoprzecinkowych w programie (lub jego fragmencie), l_o, oraz uzyskaną w obliczeniach wydajność W, czas wykonania można łatwo obliczyć jako:

$$T = \frac{l_o}{W}$$

Wartość l_o jest zależna tylko od wykonywanego kodu, nie zależy od stosowanych optymalizacji (zakładając właściwą implementację algorytmu numerycznego). Posługując się w modelu pewnym oszacowaniem lub ograniczeniem wydajności, uzyskuje się możliwość obliczenia czasu wykonania dla różnych wartości l_o (np. dla różnych przypadków danych wejściowych). Należy analizując model, zwrócić jednak uwagę czy oszacowanie wydajności jest nadal ważne, czy zmiana danych wejściowych i liczby operacji nie powoduje zmiany oszacowania wydajności wykonania.

Oszacowanie czasu wykonania programu:

$$T \geq \max(T_o, T_d)$$

można wykorzystać do oszacowania wydajności przy jego wykonaniu. W tym celu liczbę operacji zmiennoprzecinkowych programu, l_o, dzielimy przez lewą i prawą stronę nierówności co prowadzi do wzoru:

$$\frac{l_o}{T} \leq \frac{l_o}{\max(T_o, T_d)}$$

Lewa strona otrzymanej nierówności to nic innego jak rzeczywista wydajność uzyskana podczas wykonania programu, podczas kiedy prawą stronę można dalej przekształcić stosując proste zależności matematyczne:

$$W \leq \min(\frac{l_o}{T_o}, \frac{l_o}{T_d})$$

Ostatnią transformacją jest podzielenie i pomnożenie wyrażenia zawierającego czas T_d przez liczbę dostępów do danych:

$$W \leq \min(\frac{l_o}{T_o}, \frac{l_o}{l_d} \cdot \frac{l_d}{T_d})$$

W otrzymanym wzorze występują wartości wydajności związane z wykonaniem konkretnego badanego kodu. Wielkość l_o/T_o jest związana z wydajnością wykonywania operacji przez potoki przetwarzania rozkazów zmiennoprzecinkowych, l_d/T_d odpowiada przepustowości układu procesor-pamięć DRAM.

W modelu *roofline* szacuje się obie te wartości korzystając z parametrów charakteryzujących platformę obliczeniową. Każda z występujących wydajności jest ograniczana przez maksymalną wydajność dostępną na danej platformie. Dotyczy to wydajności wykonywania operacji zmiennoprzecinkowych przez potoki przetwarzania:

$$W_o = \frac{l_o}{T_o} \leq W_o^{max}$$

oraz wydajności dostępów do danych w pamięci DRAM:

$$W_d = \frac{l_d}{T_d} \leq W_d^{max}$$

Wielkość W_d^{max} w powyższym wzorze wyrażana jest w liczbie dostępów na sekundę. Częściej stosowaną wielkością określającą wydajność pamięci jest maksymalna przepustowość (*throughput*), określana także jako szerokość pasma, W_b^{max} (*bandwidth*), wyrażana w GB/s. Obie wielkości powiązane są prostym wzorem:

$$W_b^{max} = W_d^{max} * rozmiar\_danej$$

Po podstawieniu powyższych zależności uzyskuje się podstawowy wzór modelu *roofline*:

$$W \leq \min(W_o^{max}, \frac{l_o}{l_d * rozmiar\_danej} \cdot W_b^{max}) = \min(W_o^{max}, \mathrm{IA} \cdot W_b^{max})$$

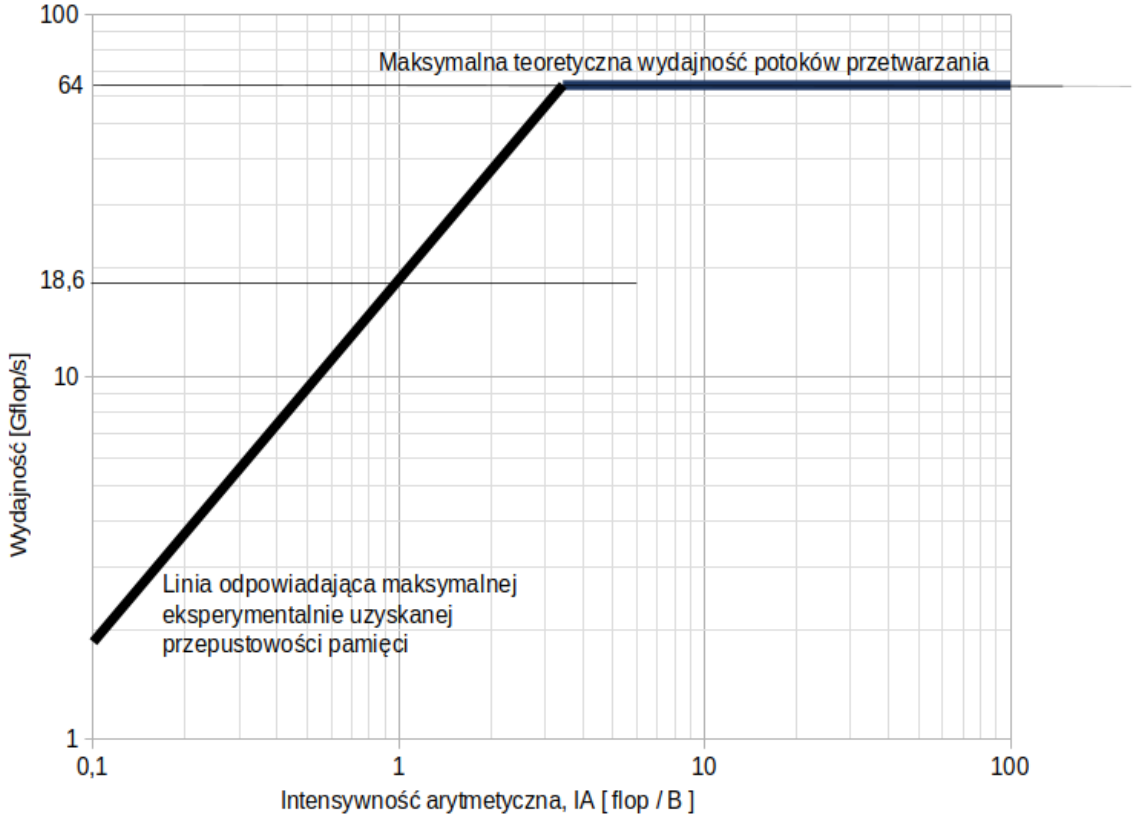
We wzorze tym występują dwa parametry charakteryzujące platformę obliczeń: W_o^{max} i W_d^{max} oraz jeden tylko parametr charakteryzujący algorytm, oznaczany dalej jako IA, określający liczbę operacji zmiennoprzecinkowych w kodzie przypadających na jeden bajt danych pobranych z pamięci DRAM:

$$\mathrm{IA} = \frac{l_o}{l_d * rozmiar\_danej} \ [\mathrm{flop/B}]$$

Parametr IA określany jest mianem intensywności arytmetycznej (*arithmetic intensity*).

Model *roofline* przedstawiany jest zazwyczaj nie w postaci wzoru, ale diagramu skonstruowanego dla konkretnej platformy. Rysunek 6.1 przedstawia podstawowy diagram modelu *roofline* dla badanego dotychczas w książce pojedynczego rdzenia mikroprocesora Intel Core i7-4790.

Diagram modelu *roofline* jest wykresem, gdzie na osi x umieszcza się intensywność arytmetyczną algorytmu, IA, a na osi y wydajność w Gflop/s. W wariancie podstawowym, na wykresie znajdują się dwie linie: jedna, pozioma, odpowiadająca maksymalnej wydajności wykonywania operacji zmiennoprzecinkowych przez potoki przetwarzania, W_o^{max}, i druga odpowiadająca maksymalnej możliwej do uzyskania wydajności w Gflop/s, przy założeniu, że wydajność jest związana wyłącznie z pobieraniem danych z pamięci DRAM (potoki przetwarzania pracują współbieżnie z pobieraniem z pamięci wykonując operacje w tle). W tym ostatnim przypadku, zgodnie z wyprowadzonymi wzorami wydajność w Gflop/s uzyskuje się poprzez pomnożenie przepustowości w GB/s przez intensywność arytmetyczną IA. Na diagramie *roofline* przypadkowi temu odpowiada linia nachylona pod kątem odpowiadającym wydajności pamięci (przepustowości), W_b^{max}. Punkty na tej linii dla konkretnych wartości na osi *x* (czyli wartości intensywności arytmetycznej IA), odpowiadają wydajności w Gflop/s równej IA*W_b^{max}.



Rysunek 6.1: Diagram modelu *roofline* dla pojedynczego rdzenia mikroprocesora Intel Core i7-4790

W przypadku diagramu dla pojedynczego rdzenia mikroprocesora Intel Core i7-4790 jako parametr W_o^{max} przyjęto wartość teoretyczną 64 Gflop/s (możliwą także do uzyskania z dużym przybliżeniem w odpowiednich testach), natomiast jako W_b^{max} użyto maksymalnej wartości z testów przepustowości z p. 4.5.3 (tabela 4.1) równą 18,6 GB/s. Wartość 18,6 GB/s prowadzi do wydajności 18,6 Gflop/s w przypadku intensywności arytmetycznej równej 1 – co zostało zaznaczone na diagramie.

Przyjęcie wartości uzyskanej eksperymentalnie jako maksymalnej przepustowości pamięci, wynika z trudności określenia maksymalnej do uzyskania w praktycznych zastosowaniach wydajności układu pamięci DRAM na podstawie samej analizy sprzętu (który w tym przypadku obejmuje mikroprocesor, płytę główną i kości pamięci DRAM). Użycie takiej określonej przez sprzęt maksymalnej teoretycznej wydajności dawałoby gwarancję, że w praktyce przepustowość nigdy nie będzie wyższa. Jednak wybór pewnej konkretnej wartości, np. podawanej przez producentów sprzętu maksymalnej teoretycznej przepustowości charakteryzującej mikroprocesor (która dla stosowanego w pracy mikroprocesora Intel Core i7-4790 podawana jest jako równa 25,6 GB/s), może prowadzić do oszacowań znacznie odbiegających od rzeczywistej możliwej do uzyskania wydajności. W przypadku wielu platform sprzętowych możliwe do uzyskania w praktyce przepustowości osiągają kilkadziesiąt procent takiego teoretycznego maksimum (wartość ok. 70% uzyskana w testach dla systemu testowego z mokroprocesorem Intel Core i7-4790 nie odbiega od charakterystyk wielu innych platform, spośród których istnieją także takie, które w praktyce nie pozwalają na osiągnięcie więcej niż 50% maksymalnej przepustowości teoretycznej).

Jak należy rozumieć diagram *roofline* dla konkretnej platformy obliczeniowej?

Przedstawia on maksymalną możliwą do uzyskania dla tej platformy wydajność w obliczeniach, gdzie dominują operacje na liczbach zmiennoprzecinkowych. Jeżeli jakiś algorytm posiada wysoką in-

tensywność arytmetyczną, dotyczy go prawa strona wykresu. W takim przypadku schemat przetwarzania, uwzględniający pełną współbieżność pracy potoków przetwarzania i układu pamięci jest taki, że po dostarczeniu pierwszej porcji danych potoki zaczynają wykonywanie operacji, a układ pamięci dostarcza kolejne porcje danych. Ze względu na to, że liczba operacji na porcji danych jest duża, układ pamięci jest w stanie dostarczyć nową porcję danych przed ukończeniem wykonywania operacji przez potoki i przejść w stan oczekiwania na nowe żądania dostępu do pamięci. Natomiast kiedy potoki przetwarzania kończą wykonywanie operacji na danych i są gotowe do przetwarzania kolejnej porcji danych, te dane już są dostarczone przez układ pamięci i potoki kontynuują pracę bez przerw. W takim przypadku czas wykonania jest wyłącznie czasem realizacji operacji przez potoki przetwarzania, a dostarczanie danych odbywa się w tle. W efekcie wydajność rzeczywista jest ograniczana wyłącznie przez wydajność potoków przetwarzania. Mówi się wtedy o wydajności ograniczanej przez możliwości procesora (*processor bound performance*).

Algorytm o niskiej intensywności arytmetycznej będzie sytuował się po lewej stronie diagramu. W jego przypadku schemat działania jest inny. Podobnie jak dla wysokiej intensywności arytmetycznej, kiedy układ pamięci dostarczy pierwszą porcję danych, potoki przetwarzania zaczynają przetwarzanie danych, a układ pamięci rozpoczyna pobieranie kolejnej porcji danych. Teraz jednak liczba operacji na jednej porcji danych jest mała, potoki przetwarzania są w stanie wykonać je, zanim układ pamięci dostarczy nowe dane. Potoki muszą czekać na dane, pozostając w stanie bezczynności. Inaczej jest z układem pamięci, który po dostarczeniu jednej porcji danych, bez konieczności oczekiwania na potoki (które w międzyczasie wygenerowały już nowe żądania dostępu do pamięci), rozpoczyna pobieranie kolejnych danych. W efekcie układ pamięci działa bez przerw, potoki przetwarzania pracują w tle, czas wykonania jest wyłącznie determinowany przez czas pobierania danych z pamięci, a wydajność ograniczana przez przepustowość układu pamięci.

Dla algorytmów o niskiej intensywności arytmetycznej potoki przetwarzania są w stanie wykonywać tylko tyle operacji, ile mają dostarczone danych. W celu uzyskania wydajności przetwarzania przez potoki w Gflop/s należy więc pomnożyć szybkość dostarczania danych (np. przepustowość wyrażaną w GB/s) przez liczbę operacji wykonywanych na porcji danych (czyli intensywność arytmetyczną wyrażaną np. w liczbie operacji na bajt). W takiej sytuacji mówi się o wydajności ograniczanej przez możliwości pamięci (*memory bound performance*).

Miejsce na osi x rozdzielające obszary wydajności ograniczanej przez możliwości procesora i wydajności ograniczanej przez możliwości układu pamięci nazywa się punktem równowagi sprzętu (*machine balance*). Dla algorytmów wykazujących taką właśnie wartość intensywności arytmetycznej możliwe (przynajmniej teoretycznie) jest wykonanie programu z potokami przetwarzania pracującymi ze swoją maksymalną wydajnością i jednocześnie z układem pamięci osiągającym swoją maksymalną przepustowość.

Dla badanej platformy pojedynczego rdzenia mikroprocesora Intel Core i7-4790 wartość *machine balance* wynosi 64/18,6≈3,44. Wartość ta wyrażana jest w jednostkach flop/B, co oznacza, że w zastosowaniu do konkretnego algorytmu wymaga uwzględnienia rozmiaru liczb używanych w obliczeniach. Dla liczb podwójnej precyzji prowadzi to do wartości 3,44*8≈27,5. Wnioskiem z tego jest, że w celu uzyskania możliwości przeprowadzania obliczeń z pełnym wykorzystaniem dostępnej wydajności potoków przetwarzania, w implementacji algorytmu powinno występować co najmniej 28 przeprowadzanych operacji arytmetycznych na każdą pobraną pojedynczą liczbę z pamięci.

Tak wysoka wartość graniczna IA jest charakterystyczna dla współczesnych architektur mikroprocesorów i prowadzi do dwóch wniosków. Z jednej strony pokazuje, że dla wielu algorytmów zdecydowanie bardziej istotna od wydajności potoków przetwarzania jest wydajność (przepustowość) pamięci DRAM. Z drugiej strony podkreśla znaczenie efektywnego wykorzystania pamięci podręcznych, które może prowadzić do redukcji transferu z pamięci DRAM i podniesienia wartości intensywności arytmetycznej dla odpowiedniej implementacji algorytmu.

6.2.1 Intensywność arytmetyczna algorytmów numerycznych

Wprowadzone na potrzeby modelu *roofline* pojęcie intensywności arytmetycznej (*arithmetic intensity*) może być powiązane z klasyfikacją podstawowych algorytmów numerycznej algebry liniowej, BLAS (*Basic Linear Algebra Subroutines*).

Tradycyjnie algorytmy BLAS dzieli się na poziomy, gdzie poziom pierwszy (*Level 1*) obejmuje algorytmy związane wyłącznie z wektorami, takie jak np. dodawanie wektorów, obliczenie normy wektora, obliczenie iloczynu skalarnego dwóch wektorów, itp. Operacje zdefiniowane w specyfikacji biblioteki BLAS uwzględniają także najczęściej możliwość przeprowadzania aktualizacji wektora będącego wynikiem operacji oraz przeskalowania, pomnożenia argumentów przez wartość skalarną. Dla operacji dodawania dwóch wektorów $\boldsymbol{x}$ i $\boldsymbol{y}$ formą podstawową jest:

$$\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$$

gdzie α jest parametrem skalarnym. Cechą istotną algorytmów BLAS poziomu pierwszego jest fakt, że liczba operacji jest zawsze rzędu rozmiaru wektorów, oznaczanego dalej przez N.

Algorytmy poziomu drugiego (*Level 2*) to operacje na wektorach i macierzach, gdzie najważniejszym algorytmem jest iloczyn macierzy i wektora. Podstawowa forma iloczynu, dla macierzy $\boldsymbol{A}$ i wektora $\boldsymbol{x}$, uwzględniająca przeskalowanie i aktualizację wektora $\boldsymbol{y}$ będącego wynikiem operacji, ma postać:

$$\boldsymbol{y} = \alpha\boldsymbol{A} \cdot \boldsymbol{x} + \beta\boldsymbol{y}$$

z dowolnymi skalarnymi parametrami α i β. Złożoność obliczeniowa algorytmów BLAS poziomu drugiego jest zawsze rzędu N^2 (w przypadku macierzy kwadratowych o wymiarze N; funkcje BLAS określone są też oczywiście dla macierzy prostokątnych).

W przypadku obu poziomów BLAS, *Level 1* i *Level 2*, liczba operacji jest tego samego rzędu co liczba dostępów do pamięci (N dla wektorów w BLAS *Level 1* i N^2 dla macierzy, i ewentualnie wielokrotnie pobieranych wektorów, w BLAS *Level 2*). Oznacza to w praktyce, że intensywność arytmetyczna algorytmów będzie zawsze mniejsza lub bliska jedności, a czas wykonania dla praktycznie wszystkich platform będzie ograniczany przez wydajność układu pamięci.

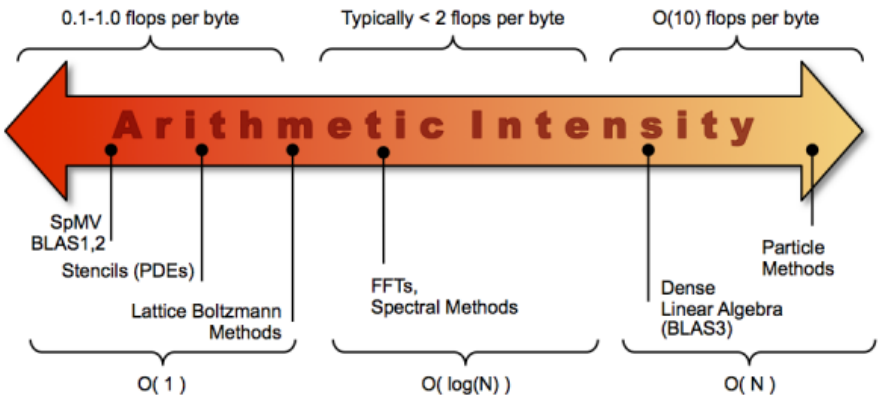
Inne charakterystyki mają algorytmy BLAS poziomu trzeciego (*Level 3*). Obejmują one szereg algorytmów, których częścią składową jest iloczyn macierzy. W postaci podstawowej, dla macierzy $\boldsymbol{A}$, $\boldsymbol{B}$ i $\boldsymbol{C}$ oraz skalarów α i β, iloczyn macierz-macierz występuje w specyfikacji BLAS jako:

$$\boldsymbol{C} = \alpha\boldsymbol{A} \cdot \boldsymbol{B} + \beta\boldsymbol{C}$$

Złożoność obliczeniowa algorytmów BLAS poziomu trzeciego jest zawsze rzędu N^3, podczas gdy złożoność pamięciowa jest rzędu N^2. Liczba dostępów, podczas realizacji algorytmów, do elementów macierzy znajdujących się w pamięci zależy jednak istotnie od implementacji i zastosowanych optymalizacji, jak to pokazują przykłady i analizy zawarte w p.5.3.2. W efekcie intensywność arytmetyczna jest zazwyczaj większa od jedności, jednak nie osiąga teoretycznie możliwego rzędu N.

Badanie typowych intensywności arytmetycznych algorytmów numerycznych można rozszerzyć poza klasyczne procedury BLAS, obejmując inne operacje, obszary zastosowań i wykorzystywane struktury danych. Rysunek 6.2 przedstawia przykładową graficzną ilustrację efektów takich badań. Algorytmy numeryczne uszeregowane są według wzrastającej intensywności arytmetycznej IA (*Arithmetic Intensity*), z zaznaczeniem teoretycznych rzędów wielkości IA (u dołu rysunku), od O(1), przez O(log(N)) do O(N), oraz praktycznie występujących w implementacjach liczb operacji zmiennoprzecinkowych na bajt pobieranych danych (dane u góry rysunku).

Uwzględnionymi w zestawieniu obszarami zastosowań, poza algorytmami BLAS, są metody: cząstek, spektralne, bazujące na szybkiej transformacji Fouriera (FFT), LBM (*Lattice Boltzmann Methods*)



Rysunek 6.2: Typowe intensywności arytmetyczne dla wybranych grup algorytmów numerycznych

oraz różnic skończonych (gdzie obliczenia opierają się na zastosowaniu "wzorców", *stencils*, do obliczeń na siatkach dyskretyzujących obszary obliczeniowe przy aproksymacji równań różniczkowych cząstkowych, *PDEs*).

Wyróżnionymi strukturami danych są macierze gęste w kontekście obliczeń BLAS Level 3 oraz macierze rzadkie. Te ostatnie często pojawiają się przy aproksymacji równań różniczkowych cząstkowych, gdzie procedury obliczeniowe obejmują rozwiązywanie układów równań liniowych, z macierzami układu będącymi macierzami rzadkimi. Jeśli przy rozwiązywaniu układów wykorzystywane są metody iteracyjne, podstawową z obliczeniowego punktu widzenia operacją staje się iloczyn macierzy rzadkiej i wektora (*SpMV, sparse matrix-vector product*). Efektywna realizacja takiego iloczynu wymaga zastosowania specjalnych struktur danych (innych od klasycznych struktur używanych w procedurach BLAS, *Level* 1 i 2), a często także istotnych modyfikacji samego algorytmu mnożenia macierz-wektor.

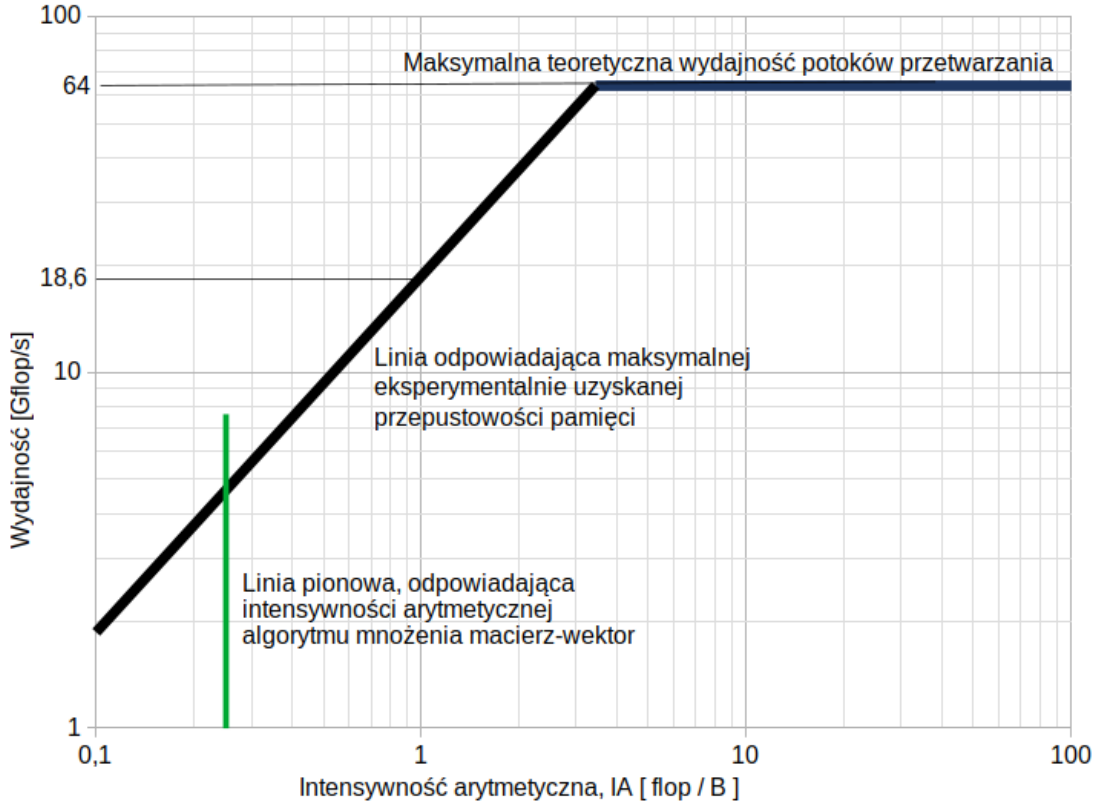
6.2.2 Przykład użycia diagramu *roofline* dla algorytmu mnożenia macierz-wektor

Podstawą wykorzystania diagramu *roofline* dla konkretnego algorytmu jest ustalenie dla tego algorytmu wartości parametru intensywności arytmetycznej, IA. W podstawowym ujęciu, IA dotyczy tylko dostępów do pamięci DRAM. Należy więc znaleźć liczbę dostępów do DRAM podczas wykonania programu oraz liczbę wykonywanych operacji zmiennoprzecinkowych.

W przypadku algorytmu mnożenia macierz-wektor, $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$, dla macierzy gęstych o rozmiarze NxN, liczba operacji wynosi $2N^2$ (N^2 mnożeń i N^2 dodawań, w analizie uwzględnia się operacje matematyczne, a nie rozkazy procesora, użycie lub nie rozkazów w rodzaju `fma`, nie zmienia wartości IA).

Analizy przeprowadzane w rozdziale 5.3.1 wskazują, że dla standardowej wersji algorytmu liczba pobrań z pamięci DRAM zmienia się od $N^2 + N$ (co oznacza jednorazowe pobranie z pamięci macierzy $\boldsymbol{A}$ oraz wektora $\boldsymbol{x}$, który w trakcie obliczeń pozostaje cały czas w pamięci podręcznej), do $2N^2$, w przypadku kiedy w każdej iteracji pętli wewnętrznej algorytmu pobierane z DRAM są jeden element macierzy i jeden element wektora (w efekcie wektor $\boldsymbol{x}$ jest pobierany N razy). W analizie pomijany jest czas zapisu N elementów wektora $\boldsymbol{y}$, będący zawsze wielkością niższego rzędu niż czas odczytów.

Przyjmując jako typ danych zmienne podwójnej precyzji o rozmiarze 8 bajtów, wartość IA, w jednostkach flop/B, zmienia się od 0,125 ($2N^2/(8 \cdot 2N^2)$) do, w przybliżeniu, 0,25 (kiedy dla dużych wartości N, N jest pomijalnie małe w stosunku do N^2). W przypadku analizowanego w pracy przykładu



Rysunek 6.3: Diagram modelu *roofline* dla pojedynczego rdzenia mikroprocesora Intel Core i7-4790 oraz algorytmu mnożenia macierz-wektor

wykonania algorytmu na pojedynczym rdzeniu mikroprocesora Intel Core i7-4790 dla N=10000, rozmiar pamięci podręcznej L3 równy 8 MB, jest wystarczający do pomieszczenia całego wektora $\boldsymbol{x}$, co prowadzi do wartości IA=0,25.

Diagram *roofline* dla konkretnego algorytmu, otrzymywany jest przez umieszczenie na diagramie dla wykorzystywanej platformy sprzętowej pionowej linii odpowiadającej wartości intensywności arytmetycznej. Rysunek 6.3 przedstawia diagram *roofline* dla algorytmu mnożenia macierz-wektor (z odpowiednio małą wartością N) oraz pojedynczego rdzenia mikroprocesora Intel Core i7-4790.

Wartość intensywności arytmetycznej 0,25 powoduje przecięcie z linią odpowiadającą wydajności ograniczanej przez pamięć, w punkcie o współrzędnej pionowej 0,25 [flop/B] * 18,6 [GB/s] = 4,65 [Gflop/s]. Jest to wartość zbliżona do uzyskiwanej w testach obliczeniowych (z dokładnością do kilku procent). Oznacza to, że program realizujący algorytm w pełni wykorzystał możliwości platformy, w tym przypadku przepustowość pamięci DRAM.

Zaletą diagramów *roofline* jest ich czytelność i względna łatwość wykorzystania w analizie wydajności, dzięki prostocie modelu. W przypadku skonstruowania diagramu *roofline* dla dowolnego algorytmu, istnieją dwa podstawowe kierunki ewentualnej dalszej optymalizacji kodu. Dla algorytmów o niskiej intensywności arytmetycznej, poniżej wartości równowagi sprzętu, dla których wydajność jest ograniczana przez pamięć, zwiększenie intensywności arytmetycznej algorytmu (np. poprzez optymalizacje *cache blocking* lub *register blocking*) może prowadzić do przesunięcia linii pionowej w prawo i w konsekwencji wyższej maksymalnej możliwej do uzyskania wartości wydajności.

Drugim kierunkiem jest uzyskanie, przy niezmienionej wartości intensywności arytmetycznej, wydajności bliższej punktowi przecięcia linii pionowej odpowiadającej wartości IA z jedną z linii ogra-

niczających wydajność w Gflop/s, charakteryzujących sprzęt. W przypadku braku możliwości zmiany intensywności arytmetycznej oraz uzyskania odpowiednio wysokiego procentu maksymalnej wydajności charakteryzującej platformę dla danej wartości IA, można uznać dalszą optymalizacje za bezcelową, ze względu na brak możliwości znaczącego zwiększenia wydajności i skrócenia czasu wykonania. Taka sytuacja ma właśnie miejsce dla rozważanej implementacji algorytmu mnożenia macierz-wektor.

Diagramy *roofline* stanowiąc istotną pomoc w analizie wydajności, maja także swoje ograniczenia. Podstawowym ograniczeniem jest duże uproszczenie modelu, co pokazuje kolejny przykład, wykorzystania diagramu *roofline* do analizy algorytmu mnożenia macierz-macierz.

6.2.3 Przykład użycia diagramu *roofline* dla algorytmu mnożenia macierz-macierz

W przypadku mnożenia macierz-macierz, jak zdarza się to często dla praktycznie stosowanych algorytmów, w szczególności algorytmów numerycznej algebry liniowej, liczba wykonywanych operacji arytmetycznych jest parametrem łatwym do ustalenia, w przeciwieństwie do liczby danych pobieranych z pamięci, która zależy od wielu czynników, co znacznie utrudnia jej określenie.

Wszystkie rozważane w pracy warianty implementacji mnożenia macierzy charakteryzują się, dla macierzy o rozmiarze NxN, liczbą operacji $2N^3$ (N^3 mnożeń i N^3 dodawań). Kwestia jakie zostają użyte w trakcie wykonania rozkazy zmiennoprzecinkowe, `mul` i `add` czy `fma`, skalarne czy wektorowe, istotna ze względu na optymalizacje wydajności, jest bez znaczenia dla obliczania wskaźnika intensywności arytmetycznej algorytmu.

Obliczenie liczby dostępów do pamięci wymaga znacznie więcej analiz i dodatkowych założeń. Pierwszym z założeń jest przyjęcie, że w celu obliczenia IA, brana jest pod uwagę rzeczywista liczba pobranych danych. Zakłada to znajomość nie tylko kodu źródłowego i postaci asemblera, ale także szczegółów wykonania kodu na konkretnym sprzęcie. Kod źródłowy i asembler wskazują na liczbę inicjowanych zapisów i odczytów danych. W rzeczywistości, każdemu zainicjowanemu dostępowi mogą towarzyszyć różne operacje realizowane przez sprzęt. Z jednej strony dostęp może zostać zrealizowany za pomocą kopii przechowywanych w pamięci podręcznej, co powoduje, że nie następuje dostęp do pamięci DRAM. Z drugiej strony dostęp do pojedynczej wartości może oznaczać podmianę całej linii pamięci podręcznej (lub kilku linii w pamięciach podręcznych różnych poziomów), czyli rzeczywisty transfer wielu zmiennych, po czym efektywnie na potrzeby algorytmu wykorzystywana jest tylko jedna zmienna, dla której inicjowany był transfer.

Przeprowadzane już wcześniej analizy (p. 5.3.2), pokazują taką właśnie rozmaitość rzeczywistego przebiegu dostępów do pamięci, wyglądających pozornie tak samo w kodzie źródłowym.

Dla implementacji naiwnej "ijk" mnożenia macierz-macierz, dla N^3 iteracji algorytmu i dwóch dostępów do pamięci inicjowanych w każdej iteracji pętli najbardziej wewnętrznej:

```
tmp = 0;
for(k=0; k<N; k++){
  tmp += A[i*N + k] * B[k*N + j]
}
C[i*N + j] = tmp;
```

realizowane jest w rzeczywistości (dla odpowiednio dużych wartości N) pobranie co najmniej $9N^3$ zmiennych (przy założeniu zmiennych podwójnej precyzji i 8 zmiennych w pojedynczej 64-bajtowej linii pamięci podręcznej). Wynika to z faktu konieczności pobrania całej nowej linii pamięci podręcznej, dla każdego pobieranego elementu macierzy $\boldsymbol{B}$. W efekcie wartość IA wynosi $2N^3/(9N^3 \cdot 8) = 2/72$ [flop/B], czyli znacznie mniej niż dla algorytmu mnożenia macierz-wektor. Maksymalna wydajność obliczeń na pojedynczym rdzeniu jest w takim przypadku ograniczana przez możliwości układu pamięci i wynosi najczęściej poniżej jednego Gflops/s (dla rdzenia Haswell w mikroprocesorze Intel Core i7-4790 ok. 0,5 Gflop/s).

Klasyczne wykorzystanie diagramu *roofline* zaleca w takim przypadku modyfikacje algorytmu prowadzące do zwiększenia wartości IA. Efekt ten uzyskuje się już po zastosowaniu wariantu "ikj" standardowego algorytmu. Wprawdzie liczba dostępów w kodzie źródłowym wręcz rośnie, trzy dostępy w każdej iteracji pętli najbardziej wewnętrznej, tym razem dwa odczyty i jeden zapis:

```
tmp = A[i*N + k];
for(j=0; j<N; j++){
  C[i*N + j] += tmp * B[k*N + j]
}
```

jednak sposób realizacji, dzięki zachowaniu lokalności przestrzennej, powoduje, że rzeczywista liczba dostępów nie odbiega od wskazywanej przez kod. Zakładając liczbę dostępów równą $3N^3$, otrzymuje się wartość IA=1/12 i maksymalną wydajność dla rdzenia mikroprocesora Intel Core i7-4790 ok. 1,5 Gflop/s.

Dalsze znaczące zwiększenie wartości IA dla algorytmu mnożenia macierzy przynosi optymalizacja *cache blocking* (p. 5.3.2). Teoretycznie liczba dostępów zmienia się do wartości rzędu $2N^3/BLS$, co prowadzi do wartości IA równej *BLS*/8 [flop/B] (dla ośmiobajtowych zmiennych podwójnej precyzji). Dla większości platform obliczeniowych oznacza to, że można dobrać rozmiar bloku tak, aby przekroczyć punkt *machine balance* i uzyskać sytuację wydajności ograniczanej przez bardzo wysoką wydajność potoków zmiennoprzecinkowych procesora (rdzenia). Warunkiem jest, aby dla danej wartości *BLS* bloki wciąż mieściły się w pamięci podręcznej. Dla pojedynczego rdzenia mikroprocesora Intel Core i7-4790 punkt równowagi sprzętu wynosi ok. 3,44 i w konsekwencji każda optymalizacja *cache blocking* z blokami o wymiarze powyżej 27 powinna prowadzić do wydajności ograniczanej przez możliwości procesora.

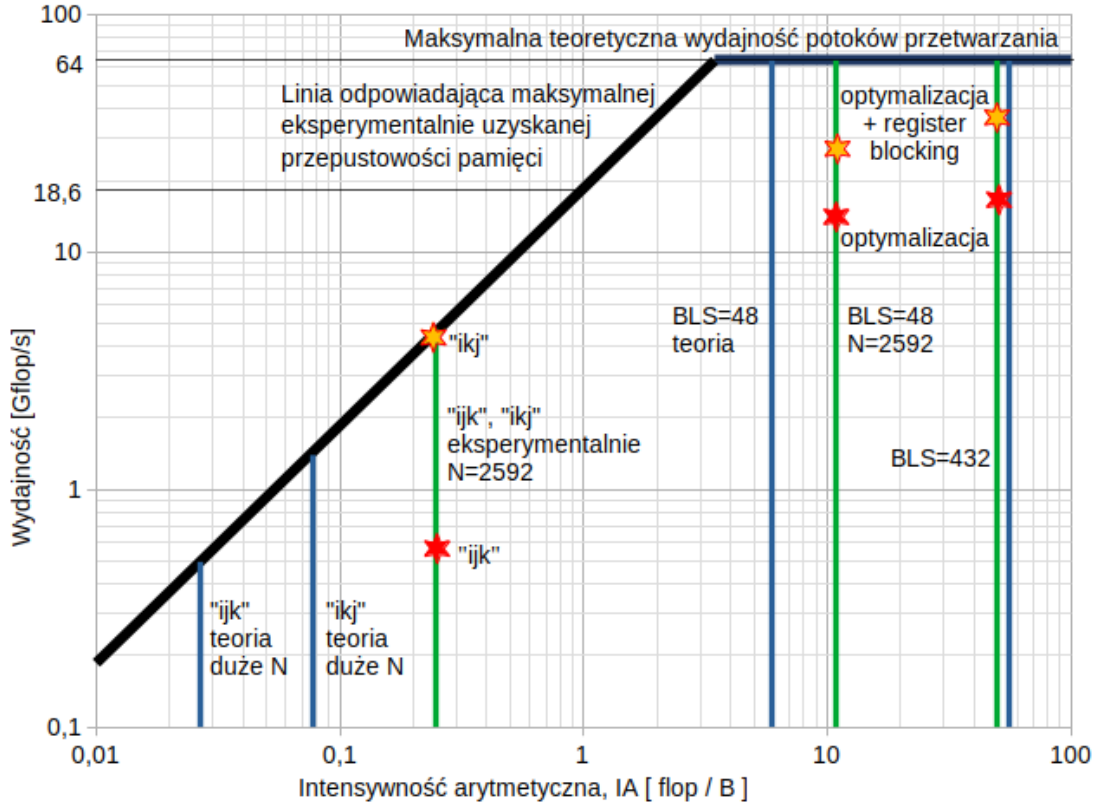
Wyniki wydajnościowe zamieszczone w pracy dotyczą optymalizacji *cache blocking* z blokami o rozmiarach 48x48 oraz 432x432, co powinno gwarantować uzyskanie wydajności ograniczanej przez procesor. W teorii pierwszy rozmiar bloków daje w wyniku IA=6, a drugi IA=54.

Założony prosty model określenia intensywności arytmetycznej IA, prowadzi często do rozbieżności z wartościami obliczanymi na podstawie rezultatów uzyskanych podczas pomiarów eksperymentalnych (tablica 5.1). W prostym modelu teoretycznym liczba dostępów określana jest na podstawie ogólnej analizy kodu, w praktyce jest obliczana na podstawie liczby chybień w pamięci podręcznej L3. W badaniach w p. 5.3.2 ta ostatnia wartość przyjmowana jest na podstawie symulacji z użyciem narzędzia *valgrind/cachegrind* (ze względu na problemy z wiarygodnością wyników zwracanych przez liczniki sprzętowe). W przypadku liczby chybień w L3 równej n_{c3}, obliczona wartość IA jest równa $2N^3/(n_{c3} \cdot 64)$.

Dla algorytmu naiwnego "ijk" obliczona wartość teoretyczna IA wynosi $2/72 \approx 0,028$. Uzyskana w symulacji *valgrind* dla wymiaru N=2592 jest znacznie wyższa $2 \cdot 2592^3/(2,2 \cdot 10^9 \cdot 64) \approx 0,25$). Przeprowadzane w p. 5.3.2 analizy wskazują na rolę odpowiedniego algorytmu podmiany linii w pamięci podręcznej, który pozwala na utrzymanie macierzy w pamięci podręcznej i uzyskanie lokalności czasowej pozwalającej na zniwelowanie wpływu braku lokalności przestrzennej[1] (efekt ten znika dla większych rozmiarów macierzy, dla których praktycznie uzyskiwana wartość IA jest zbliżona do teoretycznej).

W przypadku algorytmu "ikj", uzyskana w podstawowej analizie teoretyczna wartość IA = $1/12 \approx 0,08$ jest ok. trzykrotnie niższa od wartości wynikającej z liczby chybień w L3 zwracanej przez *cachegrind* dla obliczeń z N=2592, IA = $2 \cdot 2592^3/(2,2 \cdot 10^9 \cdot 64) \approx 0,25$ (takiej samej jak dla algorytmu "ijk"). Szczegółowe analizy z p. 5.3.2 pokazują, że przyjęta liczba dostępów $3N^3$ nie odpowiada rzeczywistemu przebiegowi obliczeń. Elementy macierzy $\boldsymbol{C}$ nie są odczytywane i zapisywane w każdej iteracji, dotyczy to tylko elementów macierzy $\boldsymbol{B}$, odczytywanych z DRAM w każdej iteracji (wystarczy do tego,

[1] Dzięki odpowiednio małemu wymiarowi N, wartość pobrana wraz z poprzedzającym elementem macierzy $\boldsymbol{B}$, mimo że z powodu braku lokalności przestrzennej nie jest użyta od razu, jednak użyta jest wystarczająco blisko w czasie, aby pozostawać do tego momentu w pamięci podręcznej (żadna linia $\boldsymbol{B}$ nie jest podmieniana w trakcie wykonania pętli wewnętrznej algorytmu).



Rysunek 6.4: Diagram modelu *roofline* dla pojedynczego rdzenia mikroprocesora Intel Core i7-4790 oraz algorytmu mnożenia macierz-macierz

aby odpowiedni wiersz macierzy $\boldsymbol{C}$ pozostawał w całości w pamięci podręcznej podczas obliczeń pętli wewnętrznej). Oznacza to, że w praktyce (oraz przy bardziej szczegółowej analizie) liczba dostępów jest równa N^3, a więc wartość IA wynosi $1/4$, tak jak pokazują eksperymenty.

Odstępstwa od przyjętego prostego modelu teoretycznego pojawiają się także dla kodu z optymalizacją *cache blocking*. Teoretyczna wartość dla bloków 48x48, IA=6 odbiega od wartości obliczonej na podstawie danych uzyskanych w obliczeniach dla N=2592, IA = $2 \cdot 2592^3/(47 \cdot 10^6 \cdot 64) \approx 11,5$. Ponownie przyczyną, wskazywaną już w p. 5.3.2, może być zaawansowany algorytm podmiany linii w pamięci podręcznej w połączeniu ze złożonym przebiegiem obliczeń (bloki 48x48 dobierane są ze względu na mieszczenie się w pamięciach L1 i L2, co może prowadzić do trudnego do przewidzenia schematu dostępów do pamięci L3).

Sytuacja staje się prostsza w przypadku bloków 432x432, dobieranych tak, aby mieściły się w pamięci L3. Prosta analiza prowadzi do wartości IA = $432/8 = 54$, natomiast wyniki eksperymentalne do wartości IA = $2 \cdot 2592^3/(11 \cdot 10^6 \cdot 64) \approx 49,5$, różniącej się od wartości teoretycznej o mniej niż 10%.

Rysunek 6.4 przedstawia diagram *roofline* dla pojedynczego rdzenia mikroprocesora Intel Core i7-4790 z naniesionymi danymi odpowiadającymi wariantom optymalizacji algorytmu mnożenia macierz-macierz. Oprócz linii pionowych dla wartości IA wynikających z podstawowej analizy kodu dla różnych wariantów optymalizacji, w przypadku odpowiednio dużych wartości N (linie niebieskie), znajdują się na nim także linie odpowiadające pogłębionej analizie wykonania oraz danym eksperymentalnym dla N=2592 (linie zielone). Dodatkowo na każdej z pionowych linii dla N=2592 zaznaczono uzyskaną eksperymentalnie wydajność obliczeń, dla kilku przypadków optymalizacji.

Eksperymentalne wartości wydajności na diagramie *roofline* 6.4 wzięte są z tabeli 5.1. Dla linii od-

powiadającej algorytmom "ijk" i "ikj", IA = 0,25, czerwona gwiazdka odpowiada wariantowi "ijk" (0,59 Gflop/s), w praktyce nie dającemu kompilatorowi możliwości efektywnej optymalizacji kodu (brak wektoryzacji) oraz prowadzącemu do dużej liczby chybień w mniejszych pamięciach L1 i L2, co nie zmienia wartości IA, zwiększa natomiast znacząco czas wykonania.

Znacznie bardziej optymistycznie wygląda sytuacja dla algorytmu "ikj", którego wydajność zaznaczona jest gwiazdką złotą. Widoczna w odpowiadającym kodzie asemblera wektoryzacja oraz uzyskane znacznie mniejsze niż w przypadku algorytmu "ijk" liczby chybień w L1 i L2, prowadzą do wydajności 4,36 Gflop/s, praktycznie na linii maksymalnej wydajności ograniczanej przez pamięć na diagramie modelu *roofline* (dla IA=0,25 mającej wartość 4,65 Gflop/s). Oznacza to, że kod jest optymalny dla danej wartości IA, a jedyną szansą jego dalszej optymalizacji jest zwiększenie wartości intensywności arytmetycznej IA.

Służąca temu celowi optymalizacja *cache blocking* przynosi zyski wydajnościowe, nie tylko w modelu teoretycznym (przesuniecie linii pionowej do wartosci IA=11,5 dla N=2592 i BLS=48), ale także w praktyce. Wprawdzie w wariancie bez optymalizacji *register blocking* wydajność osiąga tylko 14,65 Gflop/s (gwiazdka czerwona na diagramie, ok. 0,23% maksymalnej teoretycznej wydajności potoków zmiennoprzecinkowych rdzenia), jednak odpowiedni wariant *register blocking* pozwala podnieść tę wydajność do wartości 28,51 Gflop/s, gwiazdka złota na diagramie 6.4. Logarytmiczna skala na diagramie do pewnego stopnia ukrywa fakt, że wydajność ta stanowi mniej niż 50% maksymalnej teoretycznej wydajności równej 64 Gflop/s.

Starając się znaleźć sposoby zwiększenia wydajności, można, zgodnie z filozofią modelu *roofline*, dążyć do stosowania coraz większych bloków w optymalizacji *cache blocking*. W praktyce prowadzi to jednak do obniżenia wydajności, mimo wyższej wartości intensywności arytmetycznej. W tym momencie prostota podstawowego modelu *roofline* uniemożliwia znalezienie właściwego kierunku dalszych optymalizacji.

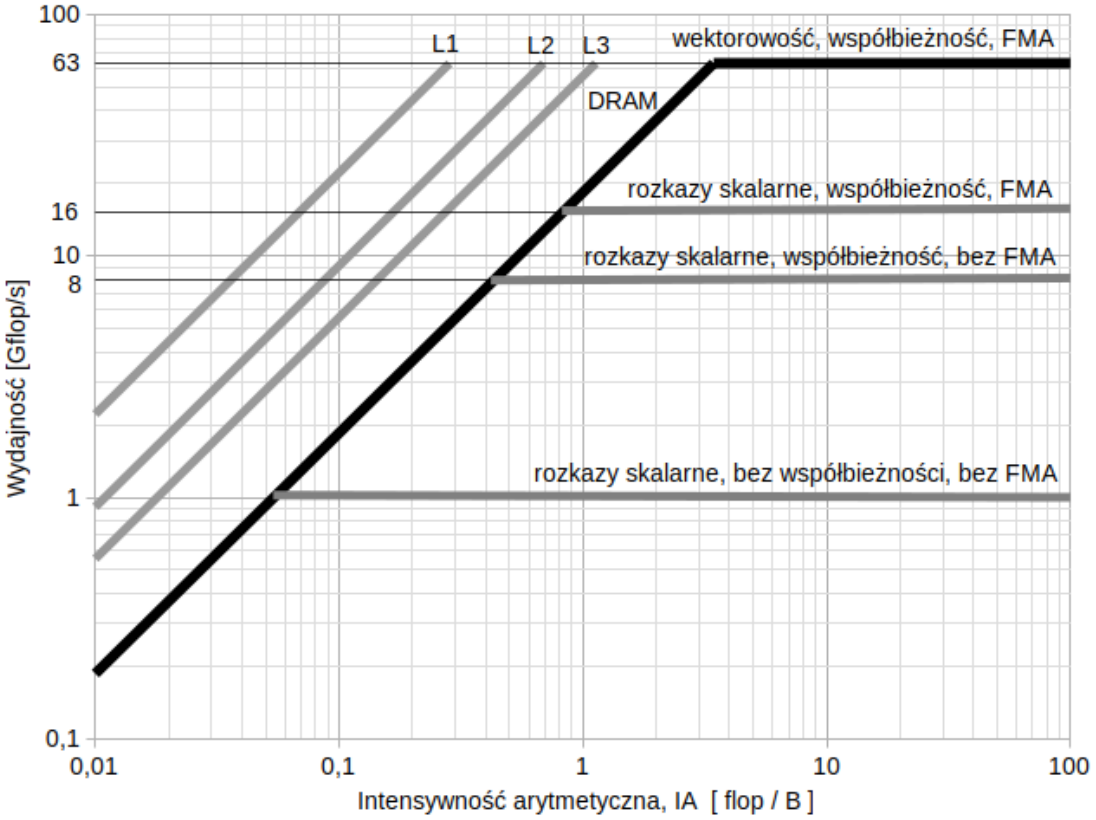
Kierunek ten daje się ustalić na podstawie bardziej szczegółowych analiz algorytmu i znajduje swoje potwierdzenie w danych tabeli 5.1. Liczba chybień w pamięci L3, będąca wskaźnikiem liczby rzeczywistych dostępów do pamięci DRAM może zostać zmniejszona, przy jednoczesnym pozostawieniu na niskim poziomie chybień w pamięci L1 i L2, poprzez zastosowanie wielopoziomowej optymalizacji *cache blocking*. Dla pojedynczego rdzenia mikroprocesora Intel Core i7-4790 takim optymalnym wariantem jest kod z dwoma wymiarami bloków BLS_1=48 i BLS_2=432.

Przesunięcie pionowej linii odpowiadającej parametrowi intensywności arytmetycznej IA, od wartości IA=11,5 dla bloków o wymiarze 48 do wartości IA=54 dla bloków o wymiarze 432, nie powoduje na diagramie *roofline* zmiany teoretycznej maksymalnej wydajności obliczeń. Jednak wydajności uzyskiwane w praktyce rosną, do 17,24 Gflop/s (ok. 27% teoretycznego maksimum, gwiazdka czerwona) dla wersji bez *register blocking*, aż do 38,16 Gflops (ok. 57% wydajności maksymalnej, gwiazdka złota) dla wersji z *register blocking*.

Otrzymane wydajności obliczeń, wziąwszy pod uwagę stopień złożoności faktycznej realizacji obliczeń przez układy mikroprocesora i systemu pamięci oraz stabilność uzyskiwanej wydajności należy uznać za sukces procesu optymalizacji. Jednak samo zastosowanie diagramu *roofline* dla rozważanego przypadku algorytmu mnożenia macierzy pozostawia otwartym pytanie dlaczego niektóre uzyskiwane w praktyce wydajności optymalizowanych implementacji pozostają relatywnie daleko od teoretycznej maksymalnej wydajności platformy, przewidywanej przez model. Odpowiedzi na to pytanie dostarczyć mogą rozszerzenia podstawowego modelu *roofline*.

6.2.4 Modyfikacje modelu *roofline*

Stosowane w praktyce rozszerzenia modelu *roofline* polegają na próbie umieszczenia na diagramie wydajności platformy dodatkowych linii odpowiadających szczegółom wykonania kodu, pomijanym w



Rysunek 6.5: Diagram rozszerzonego modelu *roofline* dla pojedynczego rdzenia mikroprocesora Intel Core i7-4790

podstawowej analizie.

Jedną grupą linii są poziome odcinki związane z wydajnością potoków zmiennoprzecinkowych w szczególnych sytuacjach, kiedy nie są wykorzystywane wszystkie możliwości optymalizacji działania potoków. Na diagramie rozszerzonego modelu *roofline* dla rdzenia mikroprocesora Intel Core i7-4790 (rys. 6.5) znajdują się trzy takie linie, oznaczone kolorem szarym.

Najwyżej położona na diagramie 6.5 pozioma linia w kolorze czarnym, identyczna z linią na oryginalnym diagramie *roofline* dla badanego rdzenia, dotyczy sytuacji wykonywania rozkazów wektorowych FMA, przez oba potoki przetwarzania rozkazów zmiennoprzecinkowych, z pełna współbieżnością.

Pierwsza dodatkowa pozioma linia, bezpośrednio poniżej linii z diagramu oryginalnego, związana jest z przypadkiem, kiedy w kodzie binarnym nie są używane rozkazy wektorowe, ale wyłącznie skalarne. Taka sytuacja może być wymuszona charakterem implementowanego algorytmu, przez co kompilator nie jest w stanie zastosować rozkazów wektorowych. Wciąż jednak stosowane są rozkazy FMA, a przetwarzanie jest w pełni współbieżne.

Kolejna linia pozioma odpowiada sytuacji przetwarzania zmiennoprzecinkowych rozkazów skalarnych z pełną współbieżnością, ale bez użycia rozkazów FMA. Ponownie może to wynikać z uwarunkowań algorytmu, uniemożliwiających kompilatorowi zastosowanie wektorowości i rozkazów FMA.

Wszystkie z trzech dotychczas omówionych linii poziomych odpowiadają przypadkom łatwo dającym się zweryfikować poprzez analizę asemblera produkowanego przez kompilator. Fakt użycia rozkazów skalarnych lub wektorowych, rozkazów FMA lub zwykłych rozkazów odpowiadających operacjom arytmetycznym, jest bezpośrednio widoczny w kodzie asemblera. Rozróżnienie pomiędzy przypadkami

nie jest jednak często możliwe na poziomie kodu źródłowego – może wynikać nie tylko z postaci kodu, ale także z użycia lub nie odpowiednich opcji kompilacji.

Ostatni, najniższy poziomy odcinek na diagramie 6.5 dotyczy przetwarzania skalarnego, bez FMA i bez współbieżności, za pomocą rozkazów skalarnych. Brak współbieżności, i w konsekwencji brak możliwości zwiększenia wydajności, może wynikać np. z istnienia nieusuwalnych zależności danych, dających się zdiagnozować i poprzez analizę asemblera, i przez badanie kodu źródłowego.

Zaznaczona na diagramie linia odpowiada przypadkowi z p. 3.10, wykonywania operacji tylko na jednej zmiennej, w sekwencji iteracji, z występującą zależnością danych pomiędzy operacjami w każdej z iteracji. Sytuacja taka może być uznana za minimalna wydajność czystego przetwarzania przez potoki zmiennoprzecinkowe pojedynczego rdzenia (bez dodatkowych spowolnień wynikających z innych typów operacji, pobierania z pamięci lub fazy pobierania i dekodowania rozkazów). Analiza sposobu przetwarzania i otrzymanych wyników prowadzi do wniosku, że decydujące o wydajności jest tutaj opóźnienie wykonywania użytych rozkazów mikroprocesora.

Użycie diagramu *roofline*, rozszerzonego o dodatkowe linie poziome, do analizy wydajności kodu wymaga szerszych działań niż w przypadku diagramu podstawowego. Nie wystarcza już tylko określenie intensywności arytmetycznej, konieczne jest także zbadanie kodu asemblera oraz kodu źródłowego. Analiza fragmentów w największym stopniu odpowiedzialnych za wydajność i czas wykonania, pozwala na określenie jaki jest są charakter realizacji programu – stopień współbieżności, wektorowość lub skalarność przetwarzania, użycie lub nie rozkazów FMA.

Ustaleniu sposobu przetwarzania przez kod daje możliwość zastanowienia się nad jego optymalnością w świetle implementowanego algorytmu oraz jego ewentualnymi modyfikacjami prowadzącymi do bardziej efektywnej realizacji. W przypadku braku możliwości takich modyfikacji diagnoza wydajności opiera się na zaznaczeniu na diagramie linii pionowej odpowiadającej intensywności arytmetycznej i punktu na linii wskazującego na uzyskaną wydajność.

Stopień optymalności obliczeń uzyskany na podstawie analizy diagramu wynika z tego jak blisko wydajność uzyskana praktycznie znajduje się w stosunku do teoretycznie najwyższej możliwej dla potoków przetwarzania w przypadku określonego sposobu funkcjonowania.

Cztery poziome linie na diagramie 6.5, z wydajnościami opisanymi na osi y, odpowiadają czterem wybranym przypadkom z tabeli 3.1 z rozdziału 3.10 (wiersze 1, 4, 8 i 12). W tabeli znajdują się wyniki uzyskane w benchmarku testującym wydajność przetwarzania potoków zmiennoprzecinkowych. Wyniki te były bardzo bliskie teoretycznym maksimom (z dokładnością do 2%). W przypadku linii poziomych na rozszerzonym diagramie *roofline*, ze względu na możliwość uzyskania w testach wydajności bardzo zbliżonej do maksymalnej, praktycznie bez znaczenia jest, czy zaznaczane są wartości uzyskane praktycznie, czy obliczone teoretycznie (na podstawie częstotliwości pracy rdzenia oraz danych sprzętowych – opóźnienia i przepustowości dla konkretnych rozkazów mikroprocesora).

Linie zaznaczone na diagramie 6.5 nie są jedynymi możliwymi do umieszczenia. Analiza algorytmu i kodu asemblera może prowadzić do wniosku, że optymalną realizacją obliczeń jest użycie rozkazów wektorowych, z pełna współbieżnością, ale bez rozkazów FMA. Ewentualnie rozkazów skalarnych FMA, ale bez pełnej współbieżności. Dla takich przypadków można obliczyć maksymalną wydajność teoretyczną potoków lub skonstruować odpowiednie benchmarki i zmierzyć praktyczne maksimum wydajności przetwarzania, a następnie porównać wydajność uzyskana w praktyce z wydajnością optymalną. Porównanie takie jest każdorazowo, wynikająca z zastosowania modelu *roofline*, wskazówką dla ewentualnej dalszej pracy nad optymalizacja kodu.

Drugą grupą linii na diagramach rozszerzonego modelu *roofline* są pochyłe linie, ponad linią wynikającą z przepustowości pamięci DRAM. Linie te określają ograniczenie wydajności przetwarzania dla algorytmów o odpowiednio niskiej intensywności arytmetycznej, dla których wydajność przetwarzania potoków wynika ze stałej szybkości dostarczania danych potokom i rośnie wraz z intensywnością arytmetyczną, czyli liczbą operacji wykonywanych na pojedynczej dostarczonej danej.

Na diagramie 6.5 trzy dodatkowe linie (oznaczone kolorem szarym) odpowiadają maksymalnym szybkościom dostarczania danych z pamięci podręcznych: L1, L2 i L3. Dla każdego z przypadków linii pochyłych (dla L1, L2, L3, a także pamięci DRAM), maksymalna wydajność potoków jest iloczynem odpowiedniej intensywności arytmetycznej i przepustowości. Na wykresie ze standardowymi skalami na osiach, większym przepustowościom odpowiadałyby linie o większym kącie nachylenia do osi x (dla danej intensywności arytmetycznej, większa przepustowość oznacza większą liczbę operacji w jednostce czasu). Ze względu na skale logarytmiczne na obu osiach diagramu *roofline*, linie proste o różnych kątach nachylenia przekształcają się w linie pochyłe o jednakowym kącie nachylenia i różnej wysokości, proporcjonalnej do współczynnika nachylenia prostej na wykresie standardowym. Linia dla L1 jest wyżej od linii dla L2, ta z kolei wyżej od linii dla L3, linia dla pamięci DRAM jest położona najniżej.

Linie na diagramie 6.5 naniesione są na podstawie testów przepustowości opisanych w p. 4.5.3. Użycie na diagramach *roofline*, dla wydajności ograniczanej przez pamięć, linii odpowiadających przepustowościom eksperymentalnym, jak było to już wcześniej wzmiankowane, jest częstą praktyką i wynika z faktu trudności w uzyskaniu maksymalnej teoretycznej przepustowości pamięci w praktycznych obliczeniach.

Każda z pochyłych linii na diagramie 6.5 reprezentuje wyniki przedstawione w tabeli 4.1, odpowiednio: 221,5 GB/s dla L1, 91,3 GB/s dla L2, 55,7 GB/s dla L3 i 18,6 GB/s dla DRAM. Wartości te można obliczyć w przybliżeniu na podstawie diagramu, np. poprzez ustalenie współrzędnej y punktu przecięcia odpowiedniej linii z osią y i pomnożenie jej przez 100 (ze względu na fakt, że oś y odpowiada intensywności arytmetycznej 0,01), .

Korzystanie z diagramu *roofline* w przypadku linii pochyłych wnosi dodatkowe komplikacje. Nie tylko konieczne jest ustalenie przepustowości, odpowiedniej dla danego poziomu w hierarchii pamięci (co w przypadku użycia ukierunkowanych na możliwości praktyczne wyników testów może być obarczone błędem), ale także określenie intensywności arytmetycznej, która jest w przypadku większości bardziej złożonych algorytmów, różna dla każdego poziomu pamięci. Liczba operacji arytmetycznych w algorytmie jest najczęściej wartością łatwą do określenia i stałą, niezależnie od implementacji i optymalizacji, natomiast objętości danych transferowane z pamięci określonych poziomów, zmieniają się w zależności od szeregu czynników. Dobrym przykładem jest tu algorytm mnożenia macierzy, gdzie istnieje wiele możliwych implementacji i dostępnych optymalizacji, manualnych oraz automatycznych, które prowadzą do zdecydowanie różnych objętości transferów dla każdego z poziomów pamięci.

Tak stworzone diagramy rozszerzonego modelu *roofline* dla konkretnego algorytmu tracą jedną ze swoich podstawowych zalet – czytelność wynikającą z prostoty modelu. Niemniej jednak bardziej szczegółowa analiza wydajności, w duchu modelu *roofline*, porównywania z możliwościami oferowanymi przez platformę obliczeniową, z uwzględnieniem wszystkich poziomów pamięci jest zadaniem ciekawym poznawczo i mogącym stanowić istotny przyczynek do optymalizacji obliczeń. Kolejny punkt przedstawia taka analizę dla przypadku algorytmu mnożenia macierzy.

6.3 Szczegółowa analiza i modelowanie wydajności dla algorytmu mnożenia macierz-macierz

Bardziej szczegółowa analiza wydajności programów prezentowana w książce oparta jest na podobnych danych jak rozszerzony model *roofline*, korzysta jednak z zestawień tabelarycznych w miejsce ilustracji graficznych. Dla wybranych operacji wykonywanych przez program konstruowana jest odpowiednia tabela służąca do przeprowadzania analiz i wnioskowania o możliwych do uzyskania wydajnościach i czasach obliczeń. Przedstawiona metoda zastosowana jest dla algorytmu mnożenia macierz-macierz, może być jednak użyta dla dowolnego algorytmu, dla którego daje się ustalić podstawowe operacje w kodzie i oszacować czasy ich realizacji.

Rodzaj operacji	Liczba operacji / rozmiar transferu	Wydajność max Gflop/s / GB/s	Minimalny czas wykonania [s]
przetwarzanie	34,83 Gflop	64 Gflop/s	0,54
transfer L1	417,94 GB	221,5 GB/s	1,89
transfer L2	16,06 GB	91,3 GB/s	0,18
transfer L3	6,02 GB	55,7 GB/s	0,11
transfer DRAM	3,01 GB	18,6 GB/s	0,16

Tablica 6.1: Tabela analizy i modelowania wydajności dla przypadku mnożenia macierzy o rozmiarze 2592x2592, realizowanego na pojedynczym rdzeniu mikroprocesora Intel Core i7-4790 za pomocą implementacji z optymalizacją *cache blocking* (bloki 48x48)

Pierwszy przykład szczegółowej analizy wydajności kodu dotyczy przypadku algorytmu mnożenia macierz-macierz, dla macierzy o wymiarze $N = 2592$, z optymalizacją *cache blocking* i rozmiarem bloków 48x48. Do przypadku tego prowadzą badania implementacji "ikj", uzyskanej jako modyfikacja (poprzez zmianę kolejności pętli) implementacji naiwnej "ijk", posługującej się nieoptymalnym wzorcem dostępu do pamięci . Mimo poprawnego wzorca, algorytm "ikj" nie osiąga wysokiej wydajności. Przyczyną jest niska intensywność arytmetyczna IA=0,25, powodująca, że wydajność implementacji jest ograniczana przez przepustowość pamięci (0,25[flop/B] * 18,6[GB/s] = 4,65[Gflop/s], co jest wartością wielokrotnie niższą od maksymalnej wydajność potoków przetwarzania równej 64 Gflop/s).

Zgodnie z zasadami optymalizacji opartej na modelu *roofline*, właściwym krokiem jest poszukiwanie implementacji o wyższej intensywności arytmetycznej. Implementacjami takimi są warianty techniki *cache blocking*, dla różnych rozmiarów bloków. Jak zostało już pokazane w p. 6.2.3, teoretycznie wystarczającym rozmiarem bloków, pozwalającym na przekroczenie wartości IA równej punktowi równowagi sprzętu, *machine balance*, i w konsekwencji na przejście do wydajności ograniczanej przez możliwości potoków rdzenia, jest 27 (dla badanej platformy sprzętowej pojedynczego rdzenia mikroprocesora Intel Core i7-4790).

Przeprowadzone eksperymenty w p. 5.3.2 (rys. 5.3) wskazują jako optymalny rozmiar bloków 48x48. W tym przypadku IA zwiększa się znacznie w stosunku do implementacji "ikj", przy czym różne metody uzyskania IA prowadzą do różnych jej wartości. Rozważania teoretyczne dają w efekcie wartość IA=6, natomiast pomiary eksperymentalne (w ramach symulacji narzędziem *cachegrind*), IA$\approx$11,5. W obu przypadkach przekroczony zostaje graniczny dla platformy punkt *machine balance*, wynoszący 3,44. W konsekwencji, model *roofline* przewiduje możliwość uzyskania, na stosowanej w obliczeniach w książce platformie testowej, wydajności ograniczonej tylko przez wydajność potoków, równej ok. 64 Gflop/s.

Uzyskiwana w praktyce wydajność poniżej 24 Gflop/s (dla kompilatora *icc*, który w tym przypadku produkował kod bardziej wydajny niż *gcc*) jest jednak znacznie niższa od teoretycznego maksimum. Wskazuje to na konieczność dalszych optymalizacji, jednak model *roofline* nie wskazuje pożądanych kierunków takich działań.

W tym momencie przeprowadzić można badania na podstawie wzmiankowanej wcześniej metody analizowania rozszerzonego zestawu operacji (poza wykonywaniem rozkazów zmiennoprzecinkowych i pobieraniem danych z pamięci DRAM), przy użyciu proponowanych zestawień tabelarycznych. Rozważanymi w niniejszym rozdziale dodatkowymi operacjami, podobnie jak w rozszerzonym modelu *roofline*, są transfery pomiędzy różnymi poziomami hierarchii pamięci.

Konstruowana na potrzeby analizy tabela zawiera wiersze dotyczące wybranych podstawowych rodzajów operacji kodu, podczas gdy w kolumnach umieszcza się parametry związane z wydajnością. Przykładowy zestaw parametrów dla każdego typu operacji, przedstawia tabela 6.1.

Pierwsza z kolumn zawierających parametry, przedstawia wartość istotną dla analizy, a jednocześnie

często trudną do określenia, liczbę operacji każdego z typów. W przypadku przetwarzania przez potoki będzie to liczba operacji zmiennoprzecinkowych, w przypadku pobierania z pamięci miarą będzie, nie liczba dostępów lub liczba podmian linii w pamięci podręcznej, ale obliczony na ich podstawie całkowity transfer z pamięci danego poziomu, wyrażany w gigabajtach. Wybór ten służy bezpośrednio obliczeniom przeprowadzanym w trakcie analizy.

Dla operacji zmiennoprzecinkowych, wiersz "przetwarzanie", liczba w tabeli jest, jak wielokrotnie było to już wskazywane w książce, łatwa do określenia. W przypadku algorytmu mnożenia macierzy kwadratowych o wymiarze N jest ona zawsze, niezależnie od implementacji, równa $2N^3$. Odpowiednią wartość dla przypadku $N = 2592$ wyrażaną w [Gflop] (miliardach operacji zmiennoprzecinkowych) zawiera odpowiednia komórka tabeli 6.1

Również relatywnie prostą do uzyskania wartością jest rozmiar danych przekazywanych między pamięcią L1 a rejestrami (wiersz "transfer L1"). Rozmiar ten obliczany jest na podstawie kodu asemblera, który jak to jest pokazane w p. 5.3.2, posiada najbardziej wewnętrzną pętlę algorytmu rozwiniętą o czynnik 4, wraz z wektoryzacją kodu. W każdej zmodyfikowanej iteracji realizowane są 3 wektorowe dostępy do pamięci (dwa odczyty i jeden zapis), każdy dotyczący 32 bajtów (transfer do i z 256-bitowych rejestrów). W związku z tym, że liczba iteracji wynosi $N^3/4$, całkowity rozmiar transferu wynosi $32 * 3 * N^3/4 = 24 * N^3$, a więc dla wymiaru $N = 2592$ prawie 418 GB.

W przypadku pamięci podręcznych bardziej odległych od potoków przetwarzania oraz pamięci DRAM, rozmiar transferu daje się ustalić tylko na podstawie pomiarów dokonywanych w trakcie wykonania. Dla rozważanego przypadku potrzebne dane zawarte są w trzecim wierszu tabeli 5.1.

Do oszacowania transferu z pamięci L2 wykorzystywana jest liczba chybień w pamięci podręcznej L1. Zakłada się, że każde chybienie w L1 oznacza przeładowanie linii o rozmiarze 64 B, która jest albo pobierana z L2, albo (w przypadku *victim cache*) pobierana z lub zapisywana do L2. W tabeli 5.1 znajdują się dwie wartości określające liczbę chybień w L1 – $149 * 10^6$ raportowane przez *valgrind/cachegrind* i $235 * 10^6$ uzyskane z odczytu liczników sprzętowych, za pomocą biblioteki PAPI (zdarzenie L1D_REPLACEMENT). Wartości różnią się znacznie, co może świadczyć o niedokładności pomiaru za pomocą PAPI lub o nieuwzględnieniu pewnych aspektów funkcjonowania rdzenia w modelu stosowanym w *cachegrind*. W tabeli, do analiz użyta jest wartość z PAPI, związana z realnym funkcjonowaniem sprzętu (mimo wszystkich możliwych zastrzeżeń do wiarygodności użycia liczników sprzętowych). Dla liczby chybień w L1 w trakcie wykonania kodu równej $235 \cdot 10^6$ rozmiar transferu wynosi 15,05 GB.

W podobny sposób szacowany jest transfer z pamięci podręcznej L3. Zakłada się, że każde chybienie w L2 oznacza pobranie linii z L3. Liczbę chybień ustala się na podstawie wskazań liczników sprzętowych dla zdarzenia L2_LINES_IN.ALL (model *cachegrind* nie uwzględnia chybień w L2). Dla $94 \cdot 10^6$ chybień oznacza to transfer 6,02 GB.

Jako ostatni szacowany jest transfer z pamięci DRAM. Tym razem dane są przyjmowane na podstawie symulacji w programie *valgrind/cachegrind*, ze względu na niską wiarygodność pomiarów za pomocą liczników sprzętowych dla zdarzeń mających miejsce poza rdzeniem mikroprocesora. Transfer pomiędzy L3 i DRAM ustalany jest jako iloczyn liczby chybień w L3 i rozmiaru linii pamięci L3. Dla $47 \cdot 10^6$ chybień daje to 3 GB.

Podstawową techniką, w prezentowanej metodzie tabelarycznej, jest szacowanie minimalnego czasu wykonania, dla poszczególnych uwzględnianych operacji. Każdorazowo uzyskaną wartość liczby operacji lub rozmiaru transferu dzieli się przez maksymalną wydajność oferowaną przez platformę obliczeniową. Odpowiada to założeniu, że transfery danych wymienione w kolumnie pierwszej tabeli, są realizowane z maksymalną przepustowością wskazaną w kolumnie drugiej.

Dla używanego w testach mnożenia macierz-macierz pojedynczego rdzenia mikroprocesora Intel Core i7-4790 maksymalne wydajności zawarte są w drugiej kolumnie tabeli 6.1. Wartością dla przetwarzania przez potoki jest maksymalna teoretyczna wydajność używanego rdzenia, 64 Gflop/s (przy

częstotliwości pracy 4 GHz). Dla transferów, dane odpowiadają wartościom uzyskanym eksperymentalnie i zawartym w tabeli 4.1 (założona jest taka sama wydajność dla odczytów i zapisów).

Kolejna kolumna tabeli przedstawia oszacowanie minimalnego czasu potrzebnego na wykonanie każdej z operacji, zakładając liczbę operacji lub rozmiar transferu z pierwszej kolumny i maksymalną wydajność z drugiej kolumny. Widać, że czasy nie rozkładają się równomiernie, trzy transfery między poziomami pamięci bardziej odległymi od potoków rdzenia wymagają znacznie mniej czasu niż wykonywanie operacji oraz transfer między L1 i rejestrami.

Pierwszym z wniosków, z tak przeprowadzonych obliczeń, jest ustalenie minimalnego czasu wykonania całego kodu. Czas taki powinien odpowiadać założeniu pełnej współbieżności podczas realizacji różnych operacji, a więc wybraniu z trzeciej kolumny w tabeli wartości najwyższej. Odpowiada ona transferowi L1-rejestry i wynosi 1,89s (w rzeczywistych eksperymentach czas wykonania jest równy 2,38s, co świadczy o relatywnie dobrym oszacowaniu, zważywszy na niezwykłą prostotę modelu).

W przypadku transferu L1-rejestry możliwe jest także użycie rozszerzonego modelu *roofline*. Obliczona na podstawie danych tabeli 6.1 wartość intensywności arytmetycznej wynosi ok. 0,08. Porównanie z danymi na diagramie rozszerzonego modelu *roofline* (rys.6.5), prowadzi do wniosku o wydajności przy wykonaniu kodu, ograniczonej przez szybkość transferu. Obliczona na podstawie diagramu wydajność, 0,08[flop/B]*221,5[GB/s]=17,72 Gflop/s, jest bliska uzyskanej w rzeczywistości wydajności 14,65 Gflop/s.

Dla pozostałych transferów między różnymi poziomami pamięci, wartości intensywności arytmetycznej (2,17 dla L2, 5,79 dla L3 i 11,57 dla DRAM) są zawsze wyższe od punktów *machine balance*, co oznacza wydajność ograniczaną przez możliwości potoków zmiennoprzecinkowych.

Drugim możliwym wnioskiem z analizy danych w tabeli 6.1 jest wskazanie kierunku dalszych optymalizacji. Standardowym sposobem działania w przypadku wielu operacji o różnych czasach wykonania, jest analiza polegająca na usuwaniu "wąskich gardeł", a więc dotyczących operacji zajmujących najwięcej czasu. Dla analizowanego przypadku jest jedna taka operacja – transfer między L1 i rejestrami.

W przypadku, kiedy przyjmuje się działanie sprzętu z maksymalną wydajnością, optymalizacja sposobu wykonywania operacji nie może prowadzić do redukcji czasu wykonania. Jedynym sposobem optymalizacji jest zmniejszenie liczby operacji, a w przypadku transferu danych, jego rozmiaru. Ten sam kierunek optymalizacji wskazywany jest przez analizę rozszerzonego modelu *roofline*. Zmniejszenie rozmiaru transferu L1-rejestry oznacza zwiększenie odpowiadającej mu wartości intensywności arytmetycznej. Celowi temu służy optymalizacja *register blocking*, której podstawowym efektem jest redukcja liczby pobrań do rejestrów i zapisów z rejestrów.

Podobnie jak w przypadku optymalizacji *cache blocking*, także dla *register blocking* trudno jest przeprowadzić rygorystyczną analizę prowadzącą do ustalenia optymalnego rozmiaru bloków. Na pewno istotne jest aby bloki były na tyle małe, żeby mieścić dane dla wielu operacji wyłącznie w rejestrach, a z drugiej strony większe bloki pozwalają na większy stopień wielokrotnego wykorzystania raz pobranych do rejestrów danych i efektywną wektoryzację.

W książce wykorzystywana jest implementacja oznaczana przez 4x12x4 (bloki 4x12 dla macierzy $\boldsymbol{B}$ i $\boldsymbol{C}$ oraz 4x4 dla macierzy $\boldsymbol{A}$). Dla takiej implementacji dane zawiera tabela 6.2. W przypadku ujętym w tabeli uwzględnione jest dodatkowo dokonanie innej optymalizacji – drugiego poziomu *cache blocking*, dla bloków 432x432. Ta ostatnia optymalizacja nie wynika z analizy danych eksperymentalnych, podyktowana jest teoretycznymi rozważaniami dotyczącymi transferu z pamięci DRAM. Rozmiar transferowanych danych i czas poświęcony na transfer mogą zostać zmniejszone, dzięki dodatkowemu zastosowaniu bloków 432x432. Ostateczna analizowana implementacja, z użyciem dwupoziomowej optymalizacji *cache blocking* (bloki 48x48 i 432x432) oraz optymalizacji *register blocking* (z wewnętrznymi pętlami rozwiniętymi o czynniki 4, 12, 4 i ręczną wektoryzacją), odpowiada ostatniemu wierszowi tabeli 5.1).

Dane w tabeli uzyskiwane są w identyczny sposób jak w przypadku z wyłącznie jednopoziomowym *cache blocking*. Dla pamięci podręcznej L1 rozmiar transferu obliczany jest na podstawie kodu asem-

Rodzaj operacji	Liczba operacji / rozmiar transferu	Wydajność max Gflop/s / GB/s	Minimalny czas wykonania [s]	Wydajność (procent max)
przetwarzanie	34,83 Gflop	64 Gflop/s	0,54	38,16 Gflop/s (60%)
transfer L1	93,60 GB	221,5 GB/s	0,42	102,85 GB/s (46%)
transfer L2	15,55 GB	91,3 GB/s	0,17	17,09 GB/s (19%)
transfer L3	6,14 GB	55,7 GB/s	0,11	6,74 GB/s (12%)
transfer DRAM	0,70 GB	18,6 GB/s	0,04	0,77 GB/s (4%)
suma czasów			0,54+0,74 = 1,28	

Tablica 6.2: Tabela analizy i modelowania wydajności dla przypadku mnożenia macierzy o rozmiarze 2592x2592 realizowanego na pojedynczym rdzeniu mikroprocesora Intel Core i7-4790 za pomocą implementacji z dwupoziomową optymalizacją *cache blocking* (bloki 48x48 i 432x432) oraz optymalizacją *register blocking* (z wewnętrznymi pętlami rozwiniętymi o czynniki 4, 12, 4 i ręczną wektoryzacją)

blera. Istotne tutaj są nie tylko dostępy do pamięci widoczne w kodzie źródłowym, ale także dodatkowe dostępy, wprowadzane przez kompilator. W analizowanym przypadku, w kodzie źródłowym optymalizacji *register blocking* z ręczną wektoryzacją w najbardziej wewnętrznej pętli wykonywanej $N^3/(4 \cdot 12 \cdot 4)$ razy znajduje się 16 skalarnych pobrań oraz 12 odczytów i 12 zapisów wektorowych, z i do pamięci. W kodzie asemblera pojawia się dodatkowo, prawdopodobnie na skutek ciśnienia na rejestry, jeszcze kilka dostępów (1 skalarny oraz 4 wektorowe). W sumie transfer wynikający z kodu źródłowego wynosi 81,28 GB, a wynikający z asemblera 93,6 GB. W tabeli, do dalszych analiz, przyjęta jest większa, odpowiadająca realnemu przebiegowi obliczeń, wartość wynikająca z asemblera.

W celu obliczenia transferu z pamięci L2 rozważane są dane z symulacji za pomocą narzędzia *valgrind/cachegrind* i wyniki odczytu liczników sprzętowych za pomocą biblioteki PAPI. Wartości liczby chybień w L1 z *cachegrind*, $190 * 10^6$, oraz liczby podmian linii w L1 z PAPI, $243 * 10^6$, różnią się o dwadzieścia parę procent. Przy przybliżonym charakterze oszacowań nie jest to różnica istotna. W tabeli, do analiz użyta jest wartość z PAPI, prowadząca do transferu 15,55 GB.

W przypadku transferu z L3 jedynym relatywnie wiarygodnym parametrem do oszacowań jest liczba pobrań linii do pamięci L2 (zdarzenie L2_LINES_IN.ALL raportowane przez PAPI). Dla $96 * 10^6$ pobranych linii transfer wynosi 6,14 GB.

Ostatnim podlegającym oszacowaniu jest transfer z pamięci DRAM. Wartość zwracana przez symulator *valgrind*, $11 * 10^6$ chybień w L3, oznacza transfer 0,7 GB. Jest to 13-krotność rozmiaru macierzy, liczba powiązana z ilorazem wymiaru macierzy przez wymiaru największego bloku, 2592/432=6. Transfer ten jest ponad czterokrotnie mniejszy od transferu dla jednopoziomowego *cache blocking* z blokami 48x48, gdzie wymiar bloku jest 9-krotnie mniejszy od przypadku 432x432.

Minimalne czasy wykonania kodu zawarte w kolumnie trzeciej tabeli, obliczone, na podstawie niezmienionych w stosunku do tabeli 6.1 maksymalnych wydajności z drugiej kolumny, pokazują, jak zmienia się charakterystyka wykonania kodu na skutek przeprowadzonych optymalizacji. Niezmienna pozostaje liczba wykonanych operacji arytmetycznych, drastycznie natomiast spada rozmiar transferu L1-rejestry. Rozmiary transferów związane z L2 i z L3 zmieniają się nieznacznie, widać także znaczącą redukcję transferu z pamięci DRAM. Wszystkie te zmiany w konsekwencji wpływają na modyfikacje minimalnych czasów realizacji operacji.

W analizowanej tabeli dla ostatecznej wersji kodu, minimalny czas realizacji operacji konkretnego rodzaju odpowiada przetwarzaniu przez potoki. Świadczy to o udanym zastosowaniu wszystkich optymalizacji *register blocking* i *cache blocking*, mających na celu zwiększenie intensywności arytmetycznej transferów do odpowiadających poziomów hierarchii pamięci. Dla każdego z transferów obliczona wartość IA (0,37 dla L1, 2,24 dla L2, 5,67 dla L3 i 49,75 dla DRAM) jest wyższa od granicznej wartości

z rozszerzonego diagramu *roofline* dla testowej platformy pojedynczego rdzenia mikroprocesora Intel Core i7-4790 (rys. 6.5).

Dane zawarte w tabeli 6.2 mogą służyć nie tylko szacowaniu minimalnego czasu wykonania poszczególnych operacji kodu, mogą także być wykorzystane do obliczenia stosunku otrzymanej wydajności wykonania danej operacji do odpowiadającej maksymalnej wydajności, zawartej w drugiej kolumnie danych w tabeli.

W tym celu dane z kolumny "liczba operacji / rozmiar transferu" dzielone są przez rzeczywisty czas wykonania, dając w wyniku uzyskaną wydajność realizacji operacji, wyrażaną odpowiednio w Gflop/s i GB/s. Z kolei tę ostatnią wartość dzieli się przez maksymalną wydajność wykonywania operacji, uzyskując odpowiedni procent ("% max"). Obie wartości dla każdej z operacji w analizowanej wersji kodu mnożenia macierz-macierz zawiera ostatnia kolumna tabeli.

Wartości w kolumnie pokazują pozytywny fakt uzyskiwania przez potoki przetwarzania ok. 60% wydajności maksymalnej, co w świetle stopnia złożoności pracy sprzętu przy wykonaniu kodu oraz uzyskiwanych wcześniej wydajności wersji niezoptymalizowanych lub zoptymalizowanych w mniejszym stopniu, należy uznać za wartość satysfakcjonującą (jak było to już wcześniej podkreślane). Jednocześnie dane tabeli wskazują na inny fakt, który dotąd nie był przedmiotem analizy. Efektem przeprowadzonych optymalizacji jest kod, w którym nie występuje jedna dominująca operacja, o czasie wykonania kilkakrotnie przekraczającym czasy realizacji pozostałych, dla której uzyskanie optymalnej wydajności staje się jedynym celem optymalizacji. Czasy wykonania operacji stają się w większym stopniu zbliżone, co może wskazywać na pracę z wyższą wydajnością poszczególnych układów sprzętowych. Dane w tabeli potwierdzają taką możliwość, w szczególności w stosunku do potoków przetwarzania operacji zmiennoprzecinkowych (60% wydajności maksymalnej) i układu pamięci podręcznej L1 (46% wydajności maksymalnej), przy najmniejszym udziale układu pamięci DRAM (4% wydajności maksymalnej).

W przypadku obu typów danych w tabeli, minimalnego szacowanego czasu wykonania i procentu rzeczywistej wydajności w stosunku do wydajności maksymalnej, uzyskane wyniki mogą być wykorzystane w optymalizacji kodu. Z jednej strony, dane mogą wskazywać na kierunki pożądanej optymalizacji – przedstawienie najdłuższych minimalnych czasów wykonania operacji prowadzi do wyznaczenia kierunków optymalizacji wąskich gardeł (przez zwiększenie wydajności realizacji danego rodzaju operacji lub zmniejszenie liczby przeprowadzanych operacji). Z drugiej strony, procent uzyskanej maksymalnej wydajności pokazuje czy podejmowanie wysiłków optymalizacji realizacji danej operacji ma sens, czy istnieje jeszcze wolna przestrzeń dla zwiększenia wydajności i skrócenia czasu wykonania.

Tabele w przedstawionej technice analizy wydajności, podobnie jak diagramy *roofline* dla konkretnych implementacji algorytmów, mogą służyć także do oszacowań całkowitego czasu wykonania programu. W dotychczasowych badaniach, dla obu technik, wykorzystywane było założenie o pełnej współbieżności realizacji operacji przez różne układy sprzętowe. Prowadziło to do szacowania minimalnego czasu wykonania kodu, przy założeniu pełnej współbieżności i maksymalnej wydajności potoków i transferów z pamięci. W praktyce stosowania metody tabelarycznej oznaczało to wybór maksymalnej wartości w kolumnie zawierającej minimalne czasy wykonania dla poszczególnych rodzajów operacji.

Uzyskany na podstawie tabeli minimalny czas wykonania kodu, czas odpowiadający przetwarzaniu przez potoki, wynosi 0,54. Rzeczywisty uzyskany czas obliczeń, 0,91s, jest znacząco dłuższy od wynikającego z analizy. W tym momencie można rozważyć hipotezę, mówiącą, że w przypadku silnie zoptymalizowanych obliczeń, kiedy nie ma pojedynczej silnie dominującej operacji i układy sprzętowe są relatywnie równomiernie obciążone, jak ma to miejsce dla badanego wariantu kodu, bardziej adekwatnym modelem mógłby być model bez założenia pełnej współbieżności wykonywania operacji.

W takiej sytuacji zamiast wzoru

$$T_m \geq \max(T_o, T_{c1}, T_{c2}, T_{c3}, T_d)$$

rozważyć można wzór

$$T_m < T_o + T_{c1} + T_{c2} + T_{c3} + T_d$$

choć ta ostatnia nierówność na pewno jest związana z błędem (sprzęt na pewno nie pracuje w pełni sekwencyjnie, choć stopień współbieżności pracy, a więc margines błędu, jest trudny do oszacowania). Dla danych z tabeli 6.2 oszacowanie czasu wykonania, odpowiadające sumie czasów poszczególnych operacji, zawiera ostatni wiersz tabeli. I tutaj różnica w stosunku do rzeczywistego czasu wykonania jest duża, $1,28/0,91 \approx 40\%$.

Analizując dane z tabeli można zwrócić uwagę na jeszcze jeden fakt. Wydajności maksymalne w wierszach odpowiadających transferom miedzy poziomami pamięci uzyskiwane są w odpowiednich benchmarkach (p. 4.5.3), w których konkretny transfer jest jedynym badanym. Badając wydajność uwzględnia się odczyty z tablicy o rozmiarze znacznie większym niż rozmiar pamięci bliższych potokom przetwarzania, ale w całości zawartej w badanej pamięci.

W celu bardziej szczegółowej analizy, rozważmy transfery z pamięci L2 i odpowiadający im benchmark. Dla stosowanej tablicy czas każdego odczytu obejmuje zawsze podmianę linii w L1 (odczyt z L2 jest konsekwencją chybienia w L1). Stąd zdarzeniem wykorzystywanym do obliczenia transferu jest L1D_REPLACEMENT. Liczba zdarzeń pomnożona przez rozmiar linii pamięci podręcznej L1 i podzielona przez czas wykonania, daje przepustowość transferów z pamięci L2. W obliczeniach pomijane są pobrania z L1 do rejestrów. Zakłada się, że rozmiar transferu z L1 do rejestrów jest identyczny z rozmiarem transferu z L2, a ze względu na wyższą przepustowość L1, transfery z L1 realizowane są w tle i nie wpływają na całkowity czas wykonania.

Analiza wykonania dla dowolnego kodu musi uwzględnić także sytuacje, kiedy rozmiar transferu z L1 jest, inaczej niż w przypadku opisywanego benchmarku, większy niż rozmiar transferu z L2 (ze względu na zasady pracy sprzętu nie może byc mniejszy). Poprawnym metodologicznie sposobem postępowania przy obliczaniu czasu wykonania w przypadku ogólnym powinno więc być uwzględnienie transferów z L2 (np. na podstawie zdarzeń L1D_REPLACEMENT) oraz transferów z L1. Jeśli jednak do obliczania czasu wykonania dla transferów z L2 jest brana przepustowość L2 z benchmarku opisywanego w p. 4.5.3, rozmiar transferu z L1, uzyskany np. z asemblera, powinien być pomniejszony o rozmiar transferu z L2 (ponieważ czasy transferów z L1 odpowiadające pobraniom z L2 są już uwzględnione w parametrze przepustowości L2).

Dotychczasowe sposoby obliczania czasu wykonania na podstawie minimalnego czasu wykonywania poszczególnych operacji, a więc przy założeniu pełnej współbieżności realizacji operacji, pozostają nadal w mocy. Uwzględnienie wyłącznie jednej operacji odpowiada sytuacjom z benchmarków, kiedy dana operacja była jedyną badaną.

Jednak w przypadku oszacowań posługujących sie sumą czasów procedurę należy zmodyfikować. Jako udział w całkowitym czasie wykonania programu dla transferów z danego poziomu pamięci podręcznej, z przepustowością uzyskana w benchmarku z p. 4.5.3 dla danego poziomu pamięci, należy przyjąć czas dla transferu z danej pamięci pomniejszony o czas dla transferu z pamięci bardziej odległej od potoków przetwarzania (czas realizacji odejmowanej części transferu, jest już uwzględniony w czasie realizacji transferu z pamięci dalszej, w efekcie zastosowania wybranej przepustowości dla pamięci dalszej). W prezentowanej w niniejszym punkcie tabeli można to uzyskać poprzez, równoważne odejmowaniu czasu, przyjecie za rozmiar transferu z danego poziomu pamięci, rozmiar obliczony jako całkowity transfer z danej pamięci pomniejszony o rozmiar transferu z kolejnej pamięci, bardziej odległej od potoków przetwarzania.

Ostatecznie do obliczenia całkowitego czasu wykonania kodu w modelu bez pełnej współbieżności wykonania wykorzystywane są:

- dla transferów z przepustowością DRAM - obliczony rozmiar transferu z DRAM

Rodzaj operacji	Liczba operacji / rozmiar transferu	Wydajność Gflop/s / GB/s	Minimalny czas wykonania [s]
przetwarzanie przez potoki	34,83 Gflop	64 Gflop/s	0,54
transfer L1 - transfer L2	78,05 GB	221,5 GB/s	0,35
transfer L2 - transfer L3	9,41 GB	91,3 GB/s	0,10
transfer L3 - transfer DRAM	5,44 GB	55,7 GB/s	0,10
transfer DRAM	0,70 GB	18,6 GB/s	0,04
suma czasów			0,54 + 0,59 = 1,13

Tablica 6.3: Tabela analizy i modelowania wydajności dla przypadku mnożenia macierzy o rozmiarze 2592x2592 realizowanego na pojedynczym rdzeniu mikroprocesora Intel Core i7-4790 za pomocą implementacji z dwupoziomową optymalizacją *cache blocking* (bloki 48x48 i 432x432) oraz optymalizacją *register blocking* (z wewnętrznymi pętlami rozwiniętymi o czynniki 4, 12, 4 i ręczną wektoryzacją) – przypadek braku współbieżności przy dostępach do pamięci różnych poziomów

- dla transferów z przepustowością L3 - obliczony rozmiar transferu z L3 pomniejszony o rozmiar transferu z DRAM
- dla transferów z przepustowością L2 - obliczony rozmiar transferu z L2 pomniejszony o rozmiar transferu z L3
- dla transferów z przepustowością L1 - obliczony rozmiar transferu z L1 pomniejszony o rozmiar transferu z L2

Jako wartości obliczone przyjmowane są dane z tabeli 6.2.

Ostatecznie, dla rozważanego przypadku algorytmu mnożenia macierz-macierz, za pomocą implementacji z dwupoziomową optymalizacją *cache blocking* (bloki 48x48 i 432x432) oraz optymalizacją *register blocking* (z wewnętrznymi pętlami rozwiniętymi o czynniki 4, 12, 4 i ręczną wektoryzacją), obliczenia czasu wykonania, zmodyfikowane zgodnie z powyższą analizą w stosunku do pierwotnej tabeli 6.2, zawiera tabela 6.3.

Dane w ostatnim wierszu tabeli pokazują, że przyjęta metodologia prowadzi do lepszego oszacowania rzeczywistego czasu wykonania (0,91s) w modelu bez współbieżności, $1,13/0,91 \approx 25\%$, niż w modelu z pełną współbieżnością, $0,54/0,91 \approx 60\%$. Potwierdza to słuszność przeprowadzonych analiz.

Jeszcze lepsze wyniki daje uwzględnienie tylko dwóch operacji, dla układów pracujących z najwyższymi wydajnościami w stosunku do swoich wydajności maksymalnych. Suma czasów wykonywania odpowiadających operacji, przez potoki przetwarzania operacji zmiennoprzecinkowych (0,54s) i układ pamięci podręcznej L1 (0,42s), wynosząca 0,96s jest z dokładnością kilku procent bliska rzeczywistemu czasowi wykonania. Trudnością przy takim sposobie postępowania jest przyjecie odpowiedniej granicy, które typy operacji uwzględnić przy obliczaniu czasu wykonania, a które pominąć.

Dotychczasowe analizy, łącznie z przedstawionym powyżej wnioskiem, o lepszym dopasowaniu modelu bez współbieżności, opierają się na założeniu, że nie ma innych czynników, który zmieniłyby postrzeganie procesu realizacji obliczeń i doprowadziły do modyfikacji wniosków z analizy wydajności. W przypadku, gdyby takie istotne inne czynniki zostały zidentyfikowane (np. nieuwzględniane dotychczas dodatkowe operacje z długim czasem wykonania, ewentualnie dodatkowe mechanizmy i parametry modyfikujące przeprowadzane obliczenia), tabele stosowane powyżej do analizy i modelowania wydajności, mogą być dalej rozszerzane i modyfikowane.

Przykładowymi dodatkowymi operacjami i mechanizmami, wspominanymi wcześniej w książce, które mogą być uwzględnione są np. wszelkie działania związane z obsługa tablicy stron w ramach

pamięci wirtualnej, pobieranie i dekodowanie rozkazów przez rdzeń, a także obsługa spekulatywnego wykonywania operacji (pobieranie z wyprzedzeniem, przewidywanie skoków). W ramach bardzo szczegółowej analizy uwzględniać można także inne, jeszcze bardziej zaawansowane aspekty funkcjonowania rdzenia mikroprocesora oraz całego układu pamięci systemu komputerowego, np. mechanizmy utrzymania spójności pamięci podręcznej, omawiane w drugiej części książki, poświęconej obliczeniom wielowątkowym i wieloprocesowym.

Rozdział 7

Ogólne wskazówki optymalizacji obliczeń sekwencyjnych (jednowątkowych)

Niniejszy krótki rozdział jest posumowaniem analiz zawartych w dotychczasowej części książki. Zbiera w jednym miejscu uwagi, które rozproszone znajdują się już w różnych miejscach tekstu, dodając kilka nowych. Opisując techniki optymalizacji odnosi się do cech sprzętu powiązanych z daną techniką, wskazując także na fragmenty książki, w których te cechy zostały wcześniej omówione.

7.1 Etapy tworzenia kodu zoptymalizowanego pod kątem wydajności

Wkomponowanie optymalizacji w proces tworzenia oprogramowania jest zagadnieniem zależnym od wielu czynników i nie da się wskazać jednego, optymalnego wzorca postępowania. Za wskazówkę w kontekście uwag zawartych w niniejszym rozdziale dotyczących tego problemu, niech posłużą słowa Donalda Knutha z Turing Award Lecture z 1974 roku: “The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (...) in programming".

Mimo tego powszechnie znanego cytatu (zwłaszcza jego ostatniej części), optymalizacja nie jest aspektem, który można odłożyć do chwili kiedy oprogramowanie jest już gotowe i spełnia wszystkie wymagania funkcjonalne. Pewne decyzje projektowe podejmowane wcześniej mogą utrudnić lub wręcz uniemożliwić optymalizację na tym końcowym etapie, bez kosztownej przebudowy kodu.

Dlatego pierwszą wskazówką jest, aby, nie poświęcając temu zagadnieniu zbyt wiele czasu i wysiłku, mieć optymalizację na względzie podczas całego procesu tworzenia oprogramowania.

Do pewnego stopnia jest to postępowanie naturalne. Jedną z pierwszych decyzji podejmowanych przy projektowaniu kodu jest ustalenie docelowej architektury sprzętu, na którym oprogramowanie będzie działać. Wybór architektury pojedynczego serwera, klastra lub sieci rozproszonej dokonywany na tym etapie, ma jak najbardziej charakter związany z wydajnością.

Po ustaleniu docelowej architektury sprzętu następuje wybór związanego z nią modelu programowania: jednowątkowe, wielowątkowe z pamięcią wspólną, wieloprocesowe z pamięcią rozproszoną, połączone ze zdalnym wywołaniem procedur lub przesyłaniem komunikatów, czy wreszcie programowanie w coraz bardziej popularnym modelu przeniesienia części obliczeń na połączone z serwerami akceleratory, np. karty graficzne (tzw. model *offload*).

Z wyborem modelu programowania związane są nie tylko organizacja kodu i jego odwzorowanie na sprzęt, ale także dobór algorytmów rozwiązujących problemy obliczeniowe. Programy często wymagają przetwarzania rozmaitych danych za pomocą jednego lub wielu algorytmów. Dobór algorytmu, spośród

wszystkich poprawnie rozwiązujących dany problem obliczeniowy, może mieć istotne znaczenie wydajnościowe. Podobnie jak wybór sprzętu, wybór algorytmu może oznaczać kilkudziesięciokrotne lub nawet kilkusetkrotne skrócenie lub wydłużenie czasu działania programu (zazwyczaj wielkość zysku lub straty zależy od egzemplarza danych wejściowych i złożoności obliczeniowej algorytmów).

Wybór algorytmu jest powiązany z architekturą sprzętu – dla danego problemu obliczeniowego inne algorytmy mogą być optymalne dla różnych modeli programowania i architektur sprzętu. Wybór pomiędzy algorytmem równoległym a sekwencyjnym musi zostać podjęty we wczesnej fazie tworzenia kodu.

Decyzja o doborze architektury sprzętu, organizacji kodu oraz stosowanych algorytmów jest często powiązana z decyzją o wykorzystaniu zewnętrznych bibliotek. Niekiedy przyczyną użycia takiej, a nie innej biblioteki, jest właśnie jej wysoka efektywność obliczeniowa. W dziedzinie obliczeń numerycznych, w szczególności w numerycznej algebrze liniowej, znane są biblioteki oferujące sprawdzone, niezawodne i wysoko wydajne procedury, dla rozmaitych modeli programowania i architektur sprzętu. Należą tu np. implementacje specyfikacji BLAS (*Basic Linear Algebra Subroutines*) i LAPACK (*Linear Algebra Package*), dla obliczeń jedno i wielowątkowych.

W każdym z modeli programowania, dla każdej z architektur sprzętu oraz każdego wybranego algorytmu, pojawia się ostatecznie etap tworzenia kodu do wykonania przez pojedynczy wątek. Temu aspektowi poświęcone są uwagi niniejszego rozdziału, aspekty związane z niektórymi innymi modelami np. z programowaniem wielowątkowym z pamięcią wspólną lub wieloprocesowym z przesyłaniem komunikatów analizowane są w dalszej części pracy.

Poświęcanie uwagi aspektom wydajnościowym w procesie tworzenia kodu jednowątkowego powinno oznaczać taką architekturę kodu, aby na tyle na ile to jest możliwe, wyodrębnić te fragmenty, które będą w największym stopniu odpowiedzialne za czas wykonania programu.

Pojęciem często używanym w tym kontekście jest *hotspot*. W różnych szczegółowych definicjach tego terminu pojawia się obraz fragmentu kodu, w którym podczas wykonywania programu realizowane jest szczególnie dużo operacji, arytmetycznych, ale także dostępów do pamięci lub urządzeń wejścia/wyjścia, zależnie od obszaru zastosowania kodu. Często nacisk położony jest na liczbę taktów zegara procesora poświęcanych na wykonywanie tego fragmentu (sumarycznych podczas wykonania całości programu – rolę *hotspot* może pełnić krótka, ale wielokrotnie wywoływana w różnych miejscach kodu funkcja).

Innym pojęciem, bardziej bezpośrednio nawiązującym do ostatecznego celu optymalizacji, czyli skrócenia czasu wykonania programu, jest wąskie gardło wydajności, *performance bottleneck*. Takim wąskim gardłem będzie fragment kodu, od którego w dużej mierze zależy całościowy czas wykonania programu. Wąskim gardłem może, ale nie musi być klasycznie definiowany *hotspot*. Może nim być także fragment rzadziej wykonywany, zawierający mniej operacji, jednak taki, że ze względu na długi czas wykonywania pojedynczej operacji (np. dostęp do twardego dysku, korzystanie z zewnętrznej bazy danych, przesyłanie komunikatu przez sieć, itp.), nawet niewielka ich ilość ma decydujący wpływ na czas wykonania.

Niezależnie od ich charakteru, istotnym na etapie tworzenia kodu jest zidentyfikowanie wąskich gardeł wydajnościowych i takie zaprojektowanie architektury programu, żeby po uzyskaniu kodu spełniającego wymagania funkcjonalne, móc przystąpić do efektywnej optymalizacji.

Pierwszym krokiem optymalizacji działającego programu, niezależnie od przeprowadzanej wstępnie identyfikacji wąskich gardeł, powinno być uzyskanie profilu wykonania. To na tym etapie następuje weryfikacja, które z fragmentów kodu rzeczywiście wpływają na czas wykonania, które zajmują jego największy procent.

Istnieje wiele narzędzi do uzyskiwania profilu wykonania kodu, jednym z najbardziej popularnych jest, wspomniany już w p.2.2, Unixowy program *grpof*. Pozwala on na uzyskiwanie, dla każdej funkcji w kodzie, czasu jej wykonania, podczas pojedynczego wywołania oraz dla całego czasu realizacji

programu. W zbiorczej formie prezentacji wyników, *flat profile*, czas podawany jest bezwzględnie, w sekundach, oraz w procencie całego czasu wykonania kodu. W wersji prezentacji wyników *call graph*, czasy uwzględniają także ”drzewo wywołań”, a więc to ile czasu zajęło wykonywanie danej funkcji, wywoływanej w konkretnym miejscu kodu, z poziomu pewnej innej funkcji.

Stosowanie programów profilujących (*profilers*) wymaga zwrócenia uwagi na kilka aspektów. Jednym z nich jest zależność czasu wykonania od rozmiaru danych wejściowych, a więc klasyczne zagadnienie złożoności obliczeniowej. Ze względu na różną złożoność różnych fragmentów kodu, inne fragmenty mogą stanowić wąskie gardła dla danych wejściowych o małych rozmiarach, a inne dla danych o rozmiarach większych.

Innym aspektem jest automatyczna optymalizacja dokonywana przez kompilatory. Inne fragmenty mogą stanowić wąskie gardła przed optymalizacją, inne po optymalizacji automatycznej. Wydajność niektórych funkcji może być znacząco poprawiana przez optymalizujące kompilatory, a innych w znacznie mniejszym stopniu lub w ogóle. Kolejną sprawą jest stosowanie wplatania w miejscu wywołania (*inlining*), dokonywane przez kompilatory. W jego efekcie niektóre funkcje z kodu źródłowego znikają podczas wykonania programu, co może utrudniać znalezienie fragmentu kodu źródłowego odpowiedzialnego za wąskie gardło wydajności.

Po zidentyfikowaniu fragmentów kodu (grup funkcji, pojedynczych funkcji lub ich części) odpowiedzialnych w największym stopniu za czas wykonania, może nastąpić ich optymalizacja. Także na tym etapie, optymalizacja może polegać po prostu na zleceniu realizacji wymagań funkcjonalnych procedurom z jednej lub kilku bibliotek. Decyzja może zależeć od istnienia odpowiednich bibliotek gwarantujących wysoką wydajność, ale także np. od wymaganej przenośności tworzonego kodu lub decyzji o unikaniu uzależniania tworzonego oprogramowania od zewnętrznych bibliotek.

Po wszystkich przygotowawczych krokach i podjęciu decyzji o manualnej optymalizacji wybranego fragmentu kodu następuje ostateczna faza pracy z kodem. Każdorazowo konieczna jest współpraca z optymalizującym kompilatorem – w książce wielokrotnie już zwracana była uwaga, że kompilatory bez włączonych opcji optymalizacji produkują programy znacznie wolniejsze niż programy zoptymalizowane. Każda wprowadzona zmiana w kodzie źródłowym powinna być testowana, najlepiej dla kilku kompilatorów i różnych opcji optymalizacji, pod kątem ewentualnych przynoszonych zysków w postaci skrócenia czasu wykonania.

Efekt konkretnych zmian w kodzie źródłowym będzie zależał i od kompilatora, i od sprzętu na którym wykonywane są obliczenia. W przypadku kiedy kompilator mimo prób odpowiedniego zapisu kodu źródłowego nie produkuje właściwego kodu binarnego, co każdorazowo można sprawdzić poprzez analizę produkowanego asemblera, jedynym rozwiązaniem może stać się samodzielne użycie rozkazów procesora, czy to poprzez zastosowanie funkcji wewnętrznych kompilatora (*compiler intrinsics*), czy samodzielne pisanie fragmentów kodu w asemblerze i łączenie z resztą kodu źródłowego.

Innym możliwym rozwiązaniem problemów z manualną optymalizacją kodu może być użycie rozmaitych języków do różnych fragmentów kodu źródłowego. Klasyczną, często stosowaną kombinacją, jest użycie języka Python do wysokopoziomowej organizacji kodu i niskopoziomowa implementacja wybranych funkcji w języku C.

7.2 Wybrane techniki optymalizacji kodu jednowątkowego ze względu na wydajność

Przegląd technik i zaleceń dotyczących optymalizacji kodu przeprowadzony jest w kolejności w przybliżeniu zgodnej z kolejnością prezentacji materiału w książce.

Pierwsze uwagi dotyczą wykonywania rozkazów składających się na kod binarny przez potoki przetwarzania rdzenia (rozdział 3. Chcąc skrócić czas wykonania programu należy dążyć do uzyskania następujących celów:

- dobór optymalnych rozkazów do uzyskania wymaganych rezultatów przez program – kiedyś zalecenie takie formułowane było jako wymaganie minimalizacji liczby rozkazów w kodzie, obecnie ze względu na istnienie różnych typów rozkazów sprawa nie jest już tak jednoznaczna. Aby ocenić stopień optymalności rozwiązania konieczna jest analiza asemblera. Na jej podstawie można próbować wyciągać wnioski o tym ile rozkazów będzie potrzebne do uzyskania wyników (np. w przypadku obliczeń numerycznych liczba rozkazów wektorowych będzie kilkukrotnie mniejsza od rozkazów skalarnych), a także jaka będzie rzeczywista przepustowość rozkazów przy wykonaniu kodu (średnia miara IPC). Ten ostatni aspekt związany jest z uwzględnieniem przetwarzania wszystkich rozkazów w sąsiedztwie czasowym każdego rozkazu, np. badając wybrane bloki podstawowe asemblera. Analiza tego aspektu prowadzi do kolejnej wskazówki.

- usuwanie zależności między rozkazami – jak jawnie pokazują to przykłady z p. 3.10, wydajność przetwarzania przez potoki istotnie zależy od występowania zależności pomiędzy wykonywanymi rozkazami. Brak takich zależności umożliwia nie tylko efektywne przetwarzanie przez każdy z potoków, ale także maksymalizuje liczbę jednocześnie wykorzystywanych potoków i pozwala rdzeniowi na stosowanie istotnej techniki wykonania poza kolejnością (*out-of-order execution*). Na poziomie kodu źródłowego niezależność rozkazów manifestuje się jako brak zależności między instrukcjami, w szczególności brak zależności danych (*data dependencies*), kiedy co najmniej dwie wykonywane instrukcje dotyczą tej samej komórki pamięci i co najmniej jedna z nich jest zapisem. Występowanie zależności jest problemem nie tylko w kontekście przetwarzania potokowego rozkazów, ale także w przypadku dowolnego przetwarzania współbieżnego, np. w programach równoległych.

- maksymalizacja liczby rozkazów przetwarzanych w jednostce czasu – zgodnie z prawem Little'a zastosowanym do przetwarzania potokowego rozkazów (p. 3.8), potoki pracują najwydajniej kiedy przetwarzają jak największa możliwą liczbę rozkazów w sposób współbieżny. Uzyskaniu tego efektu służy usuwanie zależności między rozkazami, może jednak okazać się niewystarczające. Czasem dodatkową pomocą może stać się np. gromadzenie w jednej iteracji pętli wykonywania wielu operacji arytmetycznych, co można osiągnąć np. poprzez optymalizację rozwinięcia pętli (*loop unrolling*).

- unikanie skoków – skok w kodzie asemblera każdorazowo oznacza zagrożenie płynności przetwarzania potokowego. Standardowym, związanym z tym zaleceniem jest np. eliminowanie instrukcji warunkowych i wywołań funkcji wewnątrz pętli. Sprawa staje się bardziej złożona, kiedy uwzględni się możliwości przewidywania skoków przez współczesne rdzenie mikroprocesorów. Przykład z p. 3.9.3 pokazuje, kiedy układy realizujące *branch prediction* są efektywne, a kiedy napotykają problemy. Bardziej szczegółową wskazówką powinno więc być unikanie skoków stanowiących problem dla układów przewidywania rozgałęzień.

Kolejny zestaw uwag dotyczy optymalizacji wykorzystania hierarchii pamięci współczesnych mikroprocesorów, omawianej w rozdziale 4. Z dużej liczby wskazówek, które należałoby wymienić w tym kontekście, poniżej znajduje się omówienie kilku wybranych działań:

- optymalne stosowanie lokalności odniesień w kodzie – zwiększanie lokalności odniesień, może być uznane za podstawową technikę optymalizacji w kontekście dostępów do pamięci. Obejmuje

to tak lokalność czasową, jak i przestrzenną. Każdorazowo celem jest korzystanie przy dostępach do zmiennych w kodzie źródłowym z jak najszybszej pamięci, począwszy od rejestrów, poprzez kolejne poziomy pamięci podręcznej, aż do traktowanej jako ostateczna konieczność pamięci DRAM. Do tego typu optymalizacji można także włączyć, związaną z mechanizmem pamięci wirtualnej, minimalizację liczby błędów stron, przy rozmiarach danych przekraczających pojemność pamięci DRAM (p. 4.3), w szczególności unikanie zjawiska szamotania. Innym z aspektów lokalności odniesień, wspomnianym tylko w książce, jest jego wpływ na sprawność działania pamięci wirtualnej poprzez właściwe korzystanie z tablicy stron, w szczególności unikanie chybień w pamięci podręcznej tablicy stron, TLB (p. 4.5.1). Typowe optymalizacje zmierzające do maksymalizacji lokalności odniesień (poza oczywistym unikaniem przeskoków w dostępie do elementów tablic i gromadzeniem dostępów do konkretnych zmiennych blisko siebie w kodzie) obejmują *register blocking* (p. 5.3.1) i *cache blocking* (p.5.3.2). Stosując je, w szczególności *register blocking*, należy nie dopuszczać do wystąpienia zjawiska ciśnienia na rejestry (*register pressure*)

- unikanie adresowania pośredniego tablic – przez adresowanie pośrednie w kodzie źródłowym rozumiany jest dostęp do elementu tablicy o indeksie odczytanym z innej tablicy. Tak realizowane dostępy uniemożliwiają efektywne potokowe przetwarzanie rozkazów dostępu do pamięci, co pokazują przykłady opisane w p. 4.5.1.

- unikanie nieregularnych dostępów do pamięci – technika ta ma służyć umożliwieniu zastosowania przez sprzęt efektywnego pobierania z wyprzedzeniem (*prefetching*). W przykładach z p. 4.5.1 istotnym czynnikiem umożliwiającym szybkie dostępy do pamięci było stosowanie przez sprzęt pobierania z wyprzedzeniem, możliwe do wykorzystania wtedy, gdy kolejne dostępy odbywają się zgodnie z pewnym regularnym wzorcem, możliwym do zidentyfikowania przez sprzęt i użycia do przyszłych dostępów. W praktyce może to oznaczać zastosowanie specyficznych struktur danych, np. przechowywania danych w tablicy struktur o stałych rozmiarach, zamiast np. za pomocą listy ze strukturami o zmiennym rozmiarze.

- generowanie jak największej liczby żądań dostępu do pamięci w jednostce czasu – zgodnie z prawem Little'a odniesionym do dostępów do pamięci (p. 4.5.2), potokowe jednostki odczytu i zapisu z i do pamięci pracują najefektywniej w przypadku generowania wielu żądań dostępu w sposób współbieżny. Przy tworzeniu kodu źródłowego można uzyskać taki efekt umieszczając w pojedynczej iteracji pętli dostępy do wielu tablic lub dostępy do wielu różnych miejsc pojedynczej tablicy (np. poprzez optymalizację *loop fusion* lub dowolny inny sposób organizacji kodu, taki jak np. ręczne rozwinięcie pętli). Przykład takiej pętli znajduje się w p. 4.5.3, opisującym eksperymenty mające na celu uzyskanie maksymalnej przepustowości w odczytach z pamięci.

- unikanie skoków o rozmiarach będących potęgami 2 pomiędzy kolejno odwiedzanymi elementami w tablicach, w szczególności związanych z pojedynczymi wymiarami w macierzach i tablicach wielowymiarowych (także w przypadku przechowywania w strukturach danych jako pojedynczej tablicy jednowymiarowej) – to wymaganie ma związek w praktyką wymiarowania rozmaitych elementów jednostek sprzętowych służących przechowywaniu danych jako potęg 2 (w naturalny sposób wynikające z binarności systemów komputerowych). Przyjmując za rozmiar podstawowy pojedynczy bajt, wybranymi wielokrotnościami bajtu będącymi potęgami 2 są rozmiary zmiennych, rozmiar linii pamięci podręcznej, rozmiar strony pamięci wirtualnej, rozmiary pamięci podręcznych poszczególnych poziomów. Używanie tablic o wymiarach będących potęgami 2 może przynosić korzyści wydajnościowe, jednak częściej zdarza się, że skoki miedzy kolejno odwiedzanymi elementami tablic o rozmiarze będącym potęgą 2 przynoszą straty wydajnościowe, np. ze względu na rywalizację o konkretny zbiór linii pamięci podręcznej (p. 4.4.4). Przykłady takich strat oraz optymalizacja *array padding* mająca im przeciwdziałać opisane są w p. 4.4.5.

- odpowiednie wyrównanie tablic w pamięci (p. 4.3.1) – ostatnim wspominanym aspektem optymalizacji dostępów do pamięci jest aspekt związany tylko z wysoce zoptymalizowanym kodem. Odpowiednie wyrównanie tablic (*alignment*), tak aby ich początkowe elementy zajmowały w pamięci komórki o adresach zaczynających się od konkretnej liczby bajtów (np. 64, co gwarantuje umieszczanie tablicy w pamięci podręcznej od początku pojedynczej linii) może przynieść kilkuprocentowe zyski wydajnościowe. Przyczyną wspominania o tej optymalizacji jest relatywna łatwość jej stosowania. Wystarczy użycie do dynamicznej alokacji pamięci, zamiast zwyczajowych wariantów funkcji *alloc*, specjalnych (zgodnych ze standardem POSIX) funkcji alokowania z wyrównaniem na odpowiedniej granicy.

Ostatnią omawianą grupą technik optymalizacji są techniki mające na celu efektywną współprace z kompilatorem optymalizującym. Z poziomu kodu źródłowego można próbować wspomagać pracę kompilatora mającą na celu uzyskanie kodu o minimalnym czasie wykonania poprzez m.in.:

- dobór poziomu optymalizacji, ewentualnie wybór konkretnych opcji optymalizacyjnych przy kompilacji – zastosowanie takiej strategii wymaga, najlepiej realizowanych łącznie, dokonywania przeglądu kodu asemblera i przeprowadzania eksperymentów obliczeniowych

- organizację kodu źródłowego, w tym dobór operacji i struktur danych – klasycznym przykładem tutaj jest dokonywanie ręcznie klasycznych optymalizacji opisanych w p. 5.1. Może się zdarzyć, że wprowadzenie optymalizacji klasycznych do kodu zwiększa wydajność (np. w przypadku kiedy kompilator nie zastosuje jakiejś korzystnej techniki), może być jednak tak, że spowoduje to utworzenie programu mniej wydajnego niż przed optymalizacją (jak np. w przypadku niektórych z optymalizacji opisywanych w punktach 5.3.1 i 5.3.2. Dlatego wymieniane wcześniej, przeglądanie kodu asemblera i przeprowadzanie eksperymentów obliczeniowych, są konieczne podczas dowolnych zmian wprowadzanych do kodu źródłowego pod kątem optymalizacji wydajnościowej

- użycie dyrektyw kompilatora – bywa tak, że uzyskanie odpowiedniej postaci kodu binarnego, np. z użyciem rozkazów wektorowych udaje się osiągnąć poprzez odpowiednią organizację kodu źródłowego (np. z ręczną realizacją odpowiedniego rozwinięcia pętli, jak w p. 5.3.2). Niekiedy taki cel nie zostaje osiągnięty i jedną z możliwości do wykorzystania w takiej sytuacji są jawne dyrektywy kompilatora (*compiler directives*) umieszczane bezpośrednio w kodzie. Dyrektywy takie mogą służyć np. właśnie wektoryzacji lub ułatwieniu albo wręcz wymuszeniu zrównoleglenia (co jest opisane w kolejnym rozdziale). Dyrektywy takie często były typowe dla konkretnych kompilatorów, a więc prowadziły do nieprzenośnego kodu, dopiero w ostatnich latach pojawiły się specyfikacje (np. standard OpenMP), powszechnie stosowane przez najpopularniejsze kompilatory.

Na zakończenie należy wspomnieć o efektywnej współpracy wykonywanego kodu ze środowiskiem systemu operacyjnego. Zagadnienie to w szerokim ujęciu obejmuje minimalizację wszelkiego narzutu systemowego, czasu poświęcanego na wykonywanie funkcji systemowych, takich jak alokacja pamięci, dostęp do urządzeń wejścia/wyjścia, w tym twardych dysków lub urządzeń sieciowych.

Dodatek A

Wyniki wydajnościowe obliczeń jednowątkowych dla pojedynczego rdzenia mikroprocesora Intel Core i7-9700KF o architekturze Coffee Lake

Drugim z wykorzystywanych w pracy procesorów jest 8-rdzeniowy mikroprocesor Intel Core i7-9700KF, z rdzeniami o architekturze Coffee Lake i nominalną częstotliwością pracy 3.60 GHz. Każdy rdzeń posiada dwa potoki przetwarzania rozkazów 256-bitowych. Każdy z potoków potrafi w jednym takcie kończyć jedną połączoną operację mnożenia i dodawania (*fused multiply-add – FMA*). Dla zmiennych podwójnej precyzji oznacza to 8 (4·2) skalarnych operacji kończonych w każdym takcie. Ostateczna maksymalna wydajność pojedynczego rdzenia wynosi $8 \cdot 2 \cdot 3.6 \cdot 10^9 = 57.6 \cdot 10^9$ operacji arytmetycznych podwójnej precyzji na sekundę, czyli 57.6 Gflop/s (*floating point operations per second*), co prowadzi do maksymalnej wydajności ośmiu rdzeni mikroprocesora równej 460.8 Gflop/s. W przypadku obliczeń na jednym rdzeniu możliwe jest uzyskanie częstotliwości pracy 33% wyższej od nominalnej, wynoszącej 4.8 GHz. W takim przypadku maksymalna wydajnośc pojedynczaego rdzenia osiąga 76,8 Gflop/s.

Wyniki dla testów przetwarzania potokowego rozkazów zmiennoprzecinkowych (p. 3.10)

Wyniki wydajnościowe dla testów przetwarzania potokowego z p. 3.10 zawarte są w tabeli A.1. Dla przetwarzania skalarnego w tabeli zamieszczone są wyniki otrzymane dla kompilatora *gcc* (kompilator *icc* dawał podobne wyniki), natomiast dla przetwarzania wektorowego użyto kompilatora *icc* (tutaj kompilator *gcc* nie wykorzystywał w pełni możliwości przetwarzania z wykorzystaniem wyłącznie rejestrów i prowadził do znacznie niższych wydajności).

Wersja wielowątkowa testu prowadzi do wydajności ok. 584 Gflop/s, co dla zmierzonej częstotliwości pracy ok. 4.55 GHz daje wynik zbliżony do teoretycznej maksymalnej wydajności 8x16=128 operacji na takt ([flop/cycle]).

Opis przypadku	Wydajność [Gflop/s]	Częstotliwość [GHz]	Wydajność [flop/cycle]
przetwarzanie skalarne 1-zmienna (bez FMA)	1.171	4.699	0.249
przetwarzanie skalarne 2-zmienne (bez FMA)	2.357	4.670	0.493
przetwarzanie skalarne 4-zmienne (bez FMA)	4.573	4.602	0.994
przetwarzanie skalarne 8-zmiennych (bez FMA)	9.241	4.721	1.957
przetwarzanie skalarne 10-zmiennych (bez FMA)	9.454	4.852	1.948
przetwarzanie skalarne 16-zmiennych (bez FMA)	9.439	4.729	1.996
przetwarzanie skalarne 1-zmienna (FMA)	2.330	4.699	0.496
przetwarzanie skalarne 8-zmiennych (FMA)	18.727	4.699	3.984
przetwarzanie skalarne 10-zmiennych (FMA)	18.527	4.720	3.925
przetwarzanie skalarne 16-zmiennych (FMA)	16.896	4.811	3.511
przetwarzanie wektorowe 1-tablica 4-elementowa	9.295	4.699	1.978
przetwarzanie wektorowe 1-tablica 8-elementowa	18.539	4.699	3.944
przetwarzanie wektorowe 1-tablica 16-elementowa	37.514	4.699	7.972
przetwarzanie wektorowe 2-tablice 16-elementowe	73.503	4.565	15.592
przetwarzanie wektorowe 3-tablice 16-elementowe	73.417	4.652	15.782
przetwarzanie wektorowe 4-tablice 16-elementowe	37.834	4.798	7.884

Tablica A.1: Wyniki wydajnościowe przetwarzania potokowego dla pojedynczego rdzenia mikroprocesora Intel Core i7-9700KF (opis eksperymentu w p. 3.10)

Wyniki dla testów opóźnienia i przepustowości elementów hierarchii pamięci (p. 4.5.1)

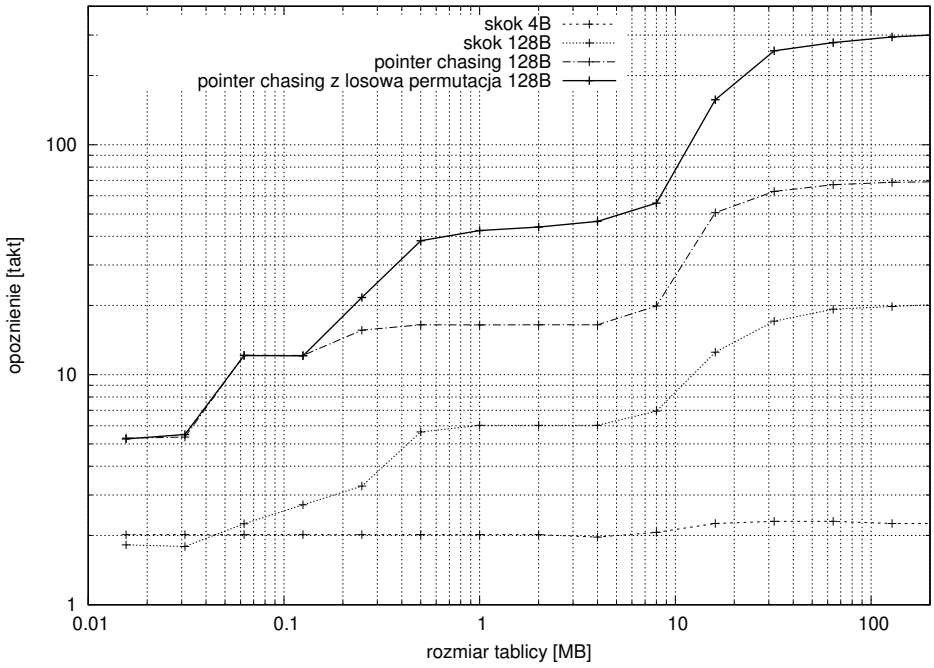
Tabelaryczne zestawienie parametrów wydajnościowych pamięci podręcznych różnych poziomów oraz pamięci DRAM dla procesora Intel Core i7-9700KF znajduje się w tabeli A.2. Rysunki A.1, A.2, A.3 oraz A.4 przedstawiają wykresy uzyskane na podstawie danych z tabeli. Charakter krzywych opóźnienia i związana z nim analiza wydajności dostępów dla różnych poziomów hierarchii pamięci są praktycznie identyczne jak w przypadku rdzenia mikroprocesora Core i7-4790 o architekturze Haswell. Wnioski z analizy, ze względu na podobieństwo obu architektur, są także analogiczne.

Wyniki wydajnościowe testów mnożenia macierz-wektor

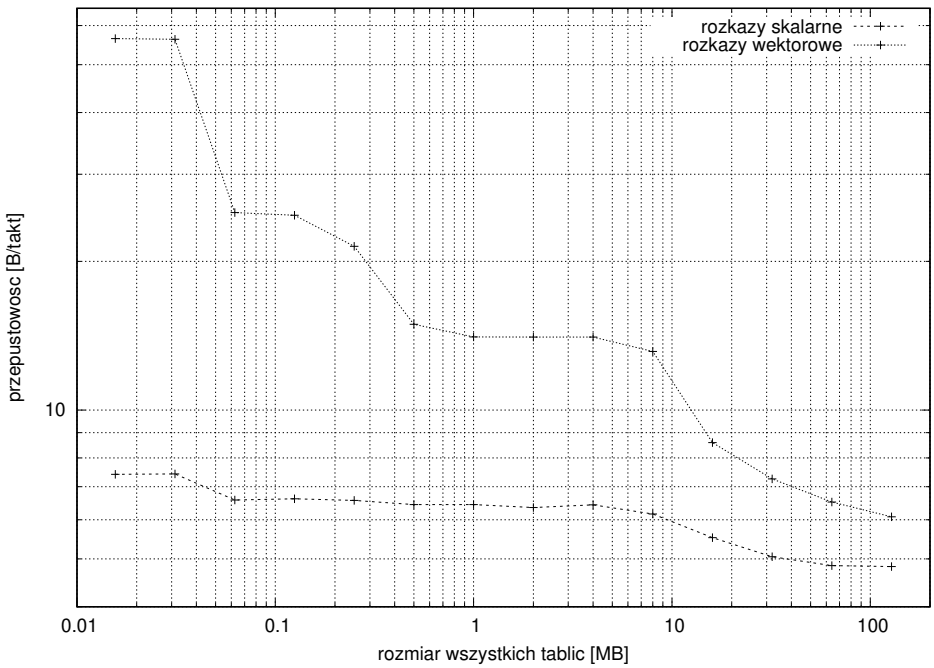
Tabela A.3 zawiera wyniki wydajnościowe testów mnożenia macierz-wektor dla przypadku macierzy 10000x10000, szczegółowo analizowanego w p. 5.3.1, uzyskane dla pojedynczego rdzenia mikroprocesora Intel Core i7-9700KF o architekturze Coffee Lake. Wyniki dotyczące wydajności przetwarzania, podawane w Gflop/s, można porównywać z parametrami zawartymi w tabeli A.1. Porównanie to ma jednak znaczenie wyłącznie poglądowe, dla algorytmu mnożenia macierz-wektor wydajność jest określana w pełni przez wydajność pobierania danych z pamięci (zgodnie z analizami w p. 5.3.1). Odpowiednie wyniki wydajnościowe podawane są w tabeli A.3 w GB/s. Należy zwrócić uwagę, że wyniki te dotyczą tylko pobierania danych macierzy z pamięci DRAM (co symbolizowane jest znakiem * w tabeli). Pomijane są dane dotyczące pobierania wektora `x` i zapisywania wektora `y`. Zgodnie z analizami z p. 5.3.1 nie powinny mieć one wpływu na czas transferu z i do pamięci głównej, ze względu na mały rozmiar i fakt, że w całości mieszczą się w pamięci L3.

Poziom w hierarchii pamięci	L1		L2		L3		DRAM	
Parametr wydajnościowy	ns	GB/s	ns	GB/s	ns	GB/s	ns	GB/s
Organizacja odczytów z pamięci								
4 tablice, liniowe, rozkazy wektorowe	0,015	270,2	0,033	121,9	0,058	69,2	0,134	29,8
4 tablice, liniowe, rozkazy skalarne	0,112	35,6	0,125	31,9	0,127	31,5	0,168	23,8
1 tablica, liniowe (skok 4 bajty)	0,42	9,5	0,42	9,6	0,42	9,6	0,47	8,4
1 tablica, liniowe (skok 128 bajtów)	0,37	10,7	0,57	7,1	1,25	3,2	4,12	1,0
j.w. + *pointer chasing*	1,12	3,58	2,53	1,58	3,43	1,17	14,11	0,28
j.w. + losowa permutacja	1,15	3,49	2,47	1,62	8,96	0,45	60,13	0,07

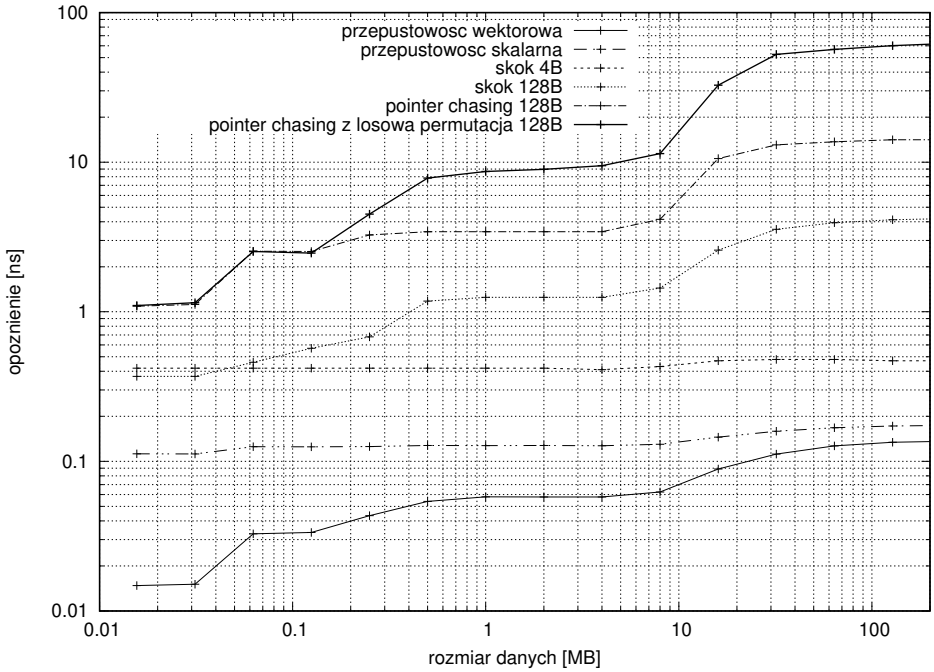
Tablica A.2: Wyniki wydajnościowe odczytów z pamięci dla jednowątkowego programu uruchomionego na rdzeniu o mikroarchitekturze Coffee Lake mikroprocesora Intel Core i7-9700KF



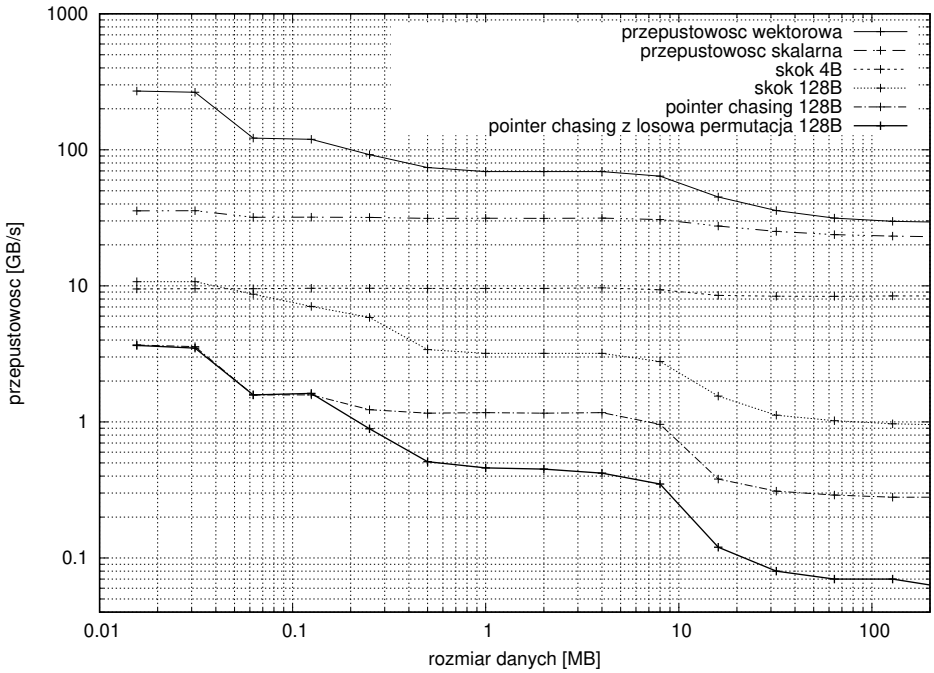
Rysunek A.1: Uzyskany eksperymentalnie czas opóźnienia przy odczycie z pamięci różnych poziomów dla pojedynczego rdzenia o mikroarchitekturze Intel Coffee Lake



Rysunek A.2: Uzyskana eksperymentalnie maksymalna przepustowość pamięci różnych poziomów dla pojedynczego rdzenia o mikroarchitekturze Intel Coffee Lake



Rysunek A.3: Zestawienie uzyskanych czasów odczytu pojedynczej zmiennej w przeprowadzonych eksperymentach dla pamięci różnych poziomów pojedynczego rdzenia o mikroarchitekturze Intel Coffee Lake



Rysunek A.4: Zestawienie uzyskanych przepustowości w przeprowadzonych eksperymentach dla pamięci różnych poziomów pojedynczego rdzenia o mikroarchitekturze Intel Coffee Lake

Niemniej dane w tabeli wskazują, że czas wykonania i uzyskana wydajność zależą nie tylko od czasu pobierania wyrazów macierzy, ale także od czasu pobierania elementów wektora `x`. Widać to z faktu, że wydajność przetwarzania rośnie dla każdego z wierszy tabeli A.3. Dla pierwszych przypadków (*gcc -O0* , *gcc -O2*, *gcc -O3*, *icc -O3*, *icc -O3 -march=core-avx2*) można wiązać to z rosnącą szybkością pobierania danych z pamięci DRAM, poprzez generowanie coraz większej liczby żądań odczytu elementów tablicy `a` w jednostce czasu (zgodnie z prawem Little'a). Jednak relatywnie znaczący wzrost wydajności, większy niż w analogicznej sytuacji dla mikroprocesora Intel Core i7-4790, dla przypadku zastosowania optymalizacji *register blocking*, świadczy o wpływie czasu pobierania elementów wektora `x` z pamięci podręcznej L3 na czas wykonania całego kodu i ostateczną uzyskaną wydajność. Kod asemblera dla tego przypadku nie różni się istotnie w stosunku do kodu bez *register blocking*, jedyną różnicą pozostaje sposób korzystania z pamięci podręcznej.

Podobnie jak dla mikroprocesora Intel Core i7-4790, także w przypadku mikroprocesora Intel Core i7-9700KF, granicą wzrostu wydajności jest czas pobierania wyrazów tablicy `a`, który nie może spaść poniżej czasu wyznaczanego przez maksymalna przepustowość pamięci DRAM. Inaczej niż dla mikroprocesora Intel Core i7-4790, wydajność pobierania zbliżoną do maksymalnej mikroprocesor Intel Core i7-9700KF uzyskuje dopiero po zastosowaniu *register blocking* (przepustowość pobierania elementów tablicy `a` równa 26,89 GB/s, podczas gdy eksperymentalnie uzyskana, zawarta w tablicy A.2, maksymalna przepustowość wynosi 29,8 GB/s). Wyniki te wskazują na konieczność przeprowadzania bardziej złożonej analizy wykonania, niż wyłącznie przetwarzanie przez potoki rdzenia i pobieranie z pamięci DRAM, w przypadku chęci uzyskania precyzyjnych oszacowań czasu wykonania kodu.

Kompilator i opcje kompilacji	Wydajność	
	[Gflop/s]	[GB/s]*
Bez register blocking		
gcc -O0	0,99	3,95
gcc -O2	2,15	8,60
gcc -O3	2,19	8,77
icc -O3	4,86	19,44
icc -O3 -march=core-avx2	5,56	22,26
Z register blocking w wersji 2x1		
icc -O3 -march=core-avx2	6,72	26,89

Tablica A.3: Wyniki wydajnościowe mnożenia macierz-wektor dla pojedynczego rdzenia mikroprocesora Intel Core i7-9700KF o architekturze Coffee Lake (opis eksperymentu w p. 5.3.1)

Implementacja	Czas [s]	Wydajność [Gflop/s]	Liczba chybień L1 [10^6]		L. chybień L2 [10^6]	L. chybień L3 [10^6]
			cgrind	PAPI	PAPI	cgrind
naiwna, ”ijk”	65.61	0.53	18 024	30 611	37 990	2 181
poprawna, ”ikj”	5.97	5.83	4 374	4 374	2 189	2 178
cache blocking - 48	1.44	24.06	149	206	167	47
cache blocking - 48+432	1.41	24.63	111	153	270	11
cache blocking - 48 + 4x12x4	0.78	44.28	189	255	169	47
c. blocking - 48+432 + 4x12x4	0.70	50.02	190	251	178	11

Tablica A.4: Parametry wykonania dla różnych implementacji mnożenia macierz-macierz na pojedynczym rdzeniu mikroprocesora Intel Core i7-9700KF

Wyniki wydajnościowe testów mnożenia macierz-macierz

”ijk” Niezwykle wysokie wartości liczby chybień w L1 i L2 raportowane przez bibliotekę PAPI, wskazują na występowanie szeregu zdarzeń sprzętowych, nie powiązanych bezpośrednio z postacią kodu, co może być wynikiem trudności w realizacji przez sprzęt, tak nieoptymalnej, a w zasadzie jak to było już określane, niepoprawnej implementacji. Prowadzi to do bardzo długiego czasu obliczeń i niskiej wydajności.

”ijk” Wyniki czasowe wskazują jednak na wzrost wydajności ponad 10-krotny. Wynika to z uporządkowania algorytmu. Wyniki dla chybień w L1 z PAPI i z *cachegrind* pokrywają się, sprzęt funkcjonuje w sposób przewidywalny.

Indeks