Analysis and modeling of

Computational Performance

Lecture 9

The anatomy of matrix-matrix product

Matrix-matrix multiplication

- Matrix multiplication ***C*** = ***A B***
  - $2n^3$ operations for square, size n, matrices
  - for infinite number of registers (or infinite cache size)
    - $3n^2$ memory accesses (or $4n^2$ if assumed reading of C)
  - optimal arithmetic intensity
    - $s_{pm} = (2n^3)/(3n^2) \sim 2n/3$
  - naive implementation -->
    - $s_{pm} = (2n^3)/(ln^3+....) \sim 2/l$
    - not only large number of memory accesses, but also accesses to b in inner loop with stride *n*
      - very low performance
- naive implementation that follows mathematical notation:
  - $c_{ij} = \Sigma_k a_{ik} b_{kj}$
- row major storage:
  - c(row, col) = c[row*n + col]

```
for(i=0;i<n;i++){
  for(j=0;j<n;j++){
    c[i*n+j]=0.0;
    for(k=0;k<n;k++){
      c[i*n+j] += a[i*n+k]*b[k*n+j];
} } }
```

Matrix-matrix multiplication

- Common sense parallel implementation
  - no apparent errors
    - initialization of c moved to separate loops
    - loop interchange for the second loop
  - optimizations left to compiler
  - still low arithmetic intensity
    - optimizations required for the three main loops

```
#pragma omp parallel default(none) \\
    shared(a,b,c,n) private(i,j,k)
{
#pragma omp for
  for(i=0;i<n;i++){
      for(j=0;j<n;j++){
         c[i*n+j]=0.0;
  }  }
#pragma omp for
  for(i=0;i<n;i++){
     for(k=0;k<n;k++){
        for(j=0;j<n;j++){
           c[i*n+j] += a[i*n+k]*b[k*n+j];
} } } }
```

Matrix-matrix multiplication

- Loop unrolling and the use of vector registers and operations

```
#pragma omp parallel for default(none) shared(a,b,c,n) private(i,j,k)
 for(int i=0; i<n; i++){
  for(int k=0; k<n; k++){
   for(int j=0; j<n; j+=4){
     //c[i*n+j] += a[i*n+k]*b[k*n+j];
     __m256d v_c11 = _mm256_load_pd(&c[i*n+j]);
     __m256d v_a11 = _mm256_broadcast_sd(&a[i*n+k]);
     __m256d v_b11 = _mm256_load_pd(&b[k*n+j]);
     v_c11 = _mm256_fmadd_pd(v_a11, v_b11, v_c11);
     __mm256_store_pd(&c[i*n+j], v_c11);
   }
  }
 }
```

Matrix-matrix multiplication

- Register blocking
  - the small inner loops manually unrolled (with factors Bi, Bj, Bk)
  - vector registers and vector operations can be used to implement inner loops – producing vector register reuse

```
for(ii=0; ii<n; ii+=Bi){
 for(kk=0; kk<n; kk+=Bk){
  for(jj=0; jj<n; jj+=Bj){

   register int i, j, k;
   for(i=ii; i<ii+Bi; i++){
    for(k=kk; k<kk+Bk; k++){
     for(j=jj; j<jj+Bj; j++){

  c[i*n+j] += a[i*n+k]*b[k*n+j];

}}}}}}
```

```
for(i=0; i<n; i+=B){
 for(k=0; k<n; k+=B){
  for(j=0; j<n; j+=B){

c[i*n+j] += a[i*n+k]*b[k*n+j] +
            a[i*n+k+1]*b[k*n+n+j] +
            a[i*n+k+2]*b[k*n+2*n+j] +
            ... ;
c[i*n+j+1] += a[i*n+k]*b[k*n+j+1] + ...
         // etc.
}}}
```

Matrix-matrix multiplication

- Cache blocking
  - loops over matrix entries are split into several levels
    - outer levels become loops over blocks

```
 const int bls=108; // to fit L1+L2 caches
#pragma omp parallel for default(none) shared(a,b,c,n,bls) private(i,j,k)
 for(i=0;i<n;i+=bls){
  for(j=0;j<n;j+=bls){
   for(k=0;k<n;k+=bls){
```

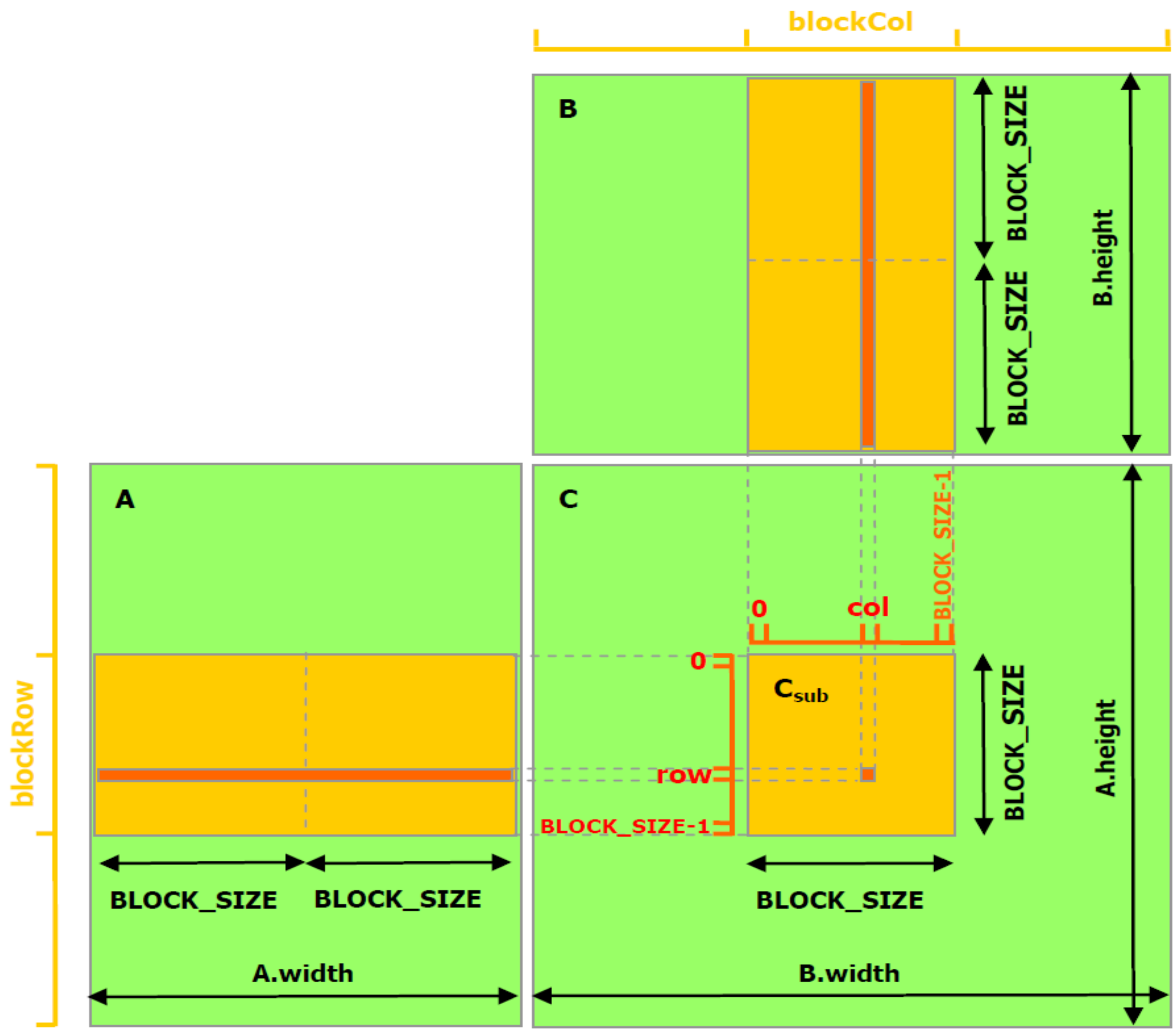
    - the innermost loops, over entries inside blocks, can be further optimized, using e.g. register blocking etc.

```
    register int kk,jj,ii;
    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] + ...
 }}}}}}
```

Matrix-matrix multiplication

One of many variants of cache blocking

- each element of block of C reused n times
- each element (row) of block of A reused BLS times
- each element (column) of block of B reused BLS times
- arithmetic intensity $s_{pm} = O((2n^3)/(2n^3/BLS))$



Matrix-matrix multiplication

- Matrix multiplication is probably the most intensively optimized function (in the world)
  - classical optimizations are usually done by compilers
  - register blocking is used to reuse registers
  - vectorization allows for using vector operations and reusing vector registers
  - cache blocking increases data locality and allows for reusing data in cache memory
    - there can be several levels of cache blocking for different levels of cache
  - more sophisticated optimizations aim at optimal usage of memory pages and TLB caches
  - the overall effect can be checked by
    - assembler inspection
    - measurements done by profilers, possibly using hardware counters
  - after all, the only thing that matters is execution time