http://wiki.cs.utexas.edu/rvdg/HowToOptimizeGemm

Based on two publications:

[1] Anatomy of high-performance matrix multiplication, Kazushige Goto, Robert A. van de Geijn, ACM Transactions on Mathematical Software (TOMS), 2008. http://www.cs.utexas.edu/~flame/web/FLAMEPublications.html. Look for Journal Publication #11

[2] BLIS: A Framework for Rapid Instantiation of BLAS Functionality., Field G. Van Zee, Robert A. van de Geijn, ACM Transactions on Mathematical Software, to appear. http://www.cs.utexas.edu/~flame/web/FLAMEPublications.html

Makefile that executes two consequtive versions of the GEMM, and plots comparison of GFLOPs for different sizes of the matrices

```
OLD  := MMult0
NEW  := MMult0
#
# sample makefile
#

CC         := gcc
LINKER     := $(CC)
CFLAGS     := -O2 -Wall -msse3
LDFLAGS    := -lm

UTIL       := copy_matrix.o \
              compare_matrices.o \
              random_matrix.o \
              dclock.o \
              REF_MMult.o \
              print_matrix.o

TEST_OBJS  := test_MMult.o $(NEW).o

%.o: %.c
        $(CC) $(CFLAGS) -c $< -o $@
%.o: %.c
        $(CC) $(CFLAGS) -c $< -o $@

all:
        make clean;
        make test_MMult.x
```

```
test_MMult.x: $(TEST_OBJS) $(UTIL) parameters.h
	$(LINKER) $(TEST_OBJS) $(UTIL) $(LDFLAGS) \
	$(BLAS_LIB) -o $(TEST_BIN) $@

run:
	make all
	echo "version = '$(NEW)';" > output_$(NEW).m
	./test_MMult.x >> output_$(NEW).m
	cp output_$(OLD).m output_old.m
	cp output_$(NEW).m output_new.m

clean:
	rm -f *.o *~ core *.x

cleanall:
	rm -f *.o *~ core *.x output*.m *.eps *.png
```

GNU-CC flags

-O2 (optimization that increases the execution time and the performance of the software)

-Wall (turns on all the warning flags - in order to have extremely clean code)

-msse3 (generates floating point operations for specific unit x86 in this case)

Parameters for the test run:

```
/*
In the test driver, there is a loop
"for ( p=PFIRST; p<= PLAST; p+= PINC )"
The below parameters set this range of values that p takes on
*/
#define PFIRST 40    // Minimal size of the matrix
#define PLAST  800   // Maximal size of the matrix
#define PINC   40    // increment step

/*
In the test driver, the m, n, and k dimensions are set to the
below values.  If the value equals "-1" then that dimension is
bound to the index p, given above.
*/

#define M -1  // bound all the three matrix dimensions
#define N -1
#define K -1

/*
```

```
In the test driver, each experiment is repeated NREPEATS times
and the best time from these repeats is used to compute the
performance
*/

#define NREPEATS 2  // each experiment is repeated 2 times
             //and the result with better performance is taken

/*
Matrices A, B, and C are stored in two dimensional arrays with
row dimensions that are greater than or equal to the row
dimension of the matrix.  This row dimension of the array is
known as the  "leading dimension" and determines the stride
(the number of double precision numbers) when one goes from one
element in a row to the next.  Having this number larger than
the row dimension of the matrix tends to adversely affect
performance.  LDX equals the leading dimension of the array
that stores matrix X.  If LDX=-1  then the leading dimension is
set to the row dimension of matrix X.
*/

#define LDA 1000 // Matrices will be stored in rows*1000 array
#define LDB 1000
#define LDC 1000
```

The C-code that is executing the particular steps of the optimized Matrix*Matrix multiplications and measures the performance

```
#include <stdio.h>
// #include <malloc.h>
#include <stdlib.h>

#include "parameters.h" // parameters for matrix sizes etc.

void REF_MMult(int, int, int, double *, int, double *, int,
double *, int );

void MY_MMult(int, int, int, double *, int, double *, int,
double *, int );

void copy_matrix(int, int, double *, int, double *, int );

void random_matrix(int, int, double *, int);

double compare_matrices( int, int, double *, int, double *, int
);

double dclock();
```

```
int main()
{
  int p, m, n, k, lda, ldb, ldc, rep;

  double dtime, dtime_best, gflops, diff;

  double *a, *b, *c, *cref, *cold;

  printf( "MY_MMult = [\n" );

  // loop through matrix sizes (from parameters.h)
  for ( p=PFIRST; p<=PLAST; p+=PINC ){
    m = ( M == -1 ? p : M );
    n = ( N == -1 ? p : N );
    k = ( K == -1 ? p : K );

    // theoretical number of GFLOPs for ideal M*M
    gflops = 2.0 * m * n * k * 1.0e-09;

    lda = ( LDA == -1 ? m : LDA );
    ldb = ( LDB == -1 ? k : LDB );
    ldc = ( LDC == -1 ? m : LDC );

    /* Allocate space for the matrices */
    /* Note: I create an extra column in A to make sure that
       prefetching beyond the matrix does not cause a segfault
*/
// we allocate space for random matrices A,B, Cold and Cref
    a = ( double * ) malloc( lda * (k+1) * sizeof( double ) );
    b = ( double * ) malloc( ldb * n * sizeof( double ) );
    c = ( double * ) malloc( ldc * n * sizeof( double ) );
    cold = ( double * ) malloc( ldc * n * sizeof( double ) );
    cref = ( double * ) malloc( ldc * n * sizeof( double ) );

    /* Generate random matrices A, B, Cold */
// lda is how many columns we keep in 2d array storing matrix
row by row
// we have C=A*B mxn = mxk*kxn (and the multiplication eats k)
// allocate matrix A m x k
    random_matrix( m, k, a, lda );
// allocate matrix B k x n
    random_matrix( k, n, b, ldb );
// allocate matrix Cold m x n
    random_matrix( m, n, cold, ldc );

// copy matrix cold to cref
    copy_matrix( m, n, cold, ldc, cref, ldc );

// now we have random matrices A,B, Cold and Cref
```

```
    /* Run the reference implementation so the answers can be
compared */

// this just generates matrix Cref using "unoptimized" routine
// so we can compare the matrices resulting from the
execution // of optimized and unoptimized routines and make
sure
// that both are identical
// (that our optimization didn't damage the M*M algorithm)
    REF_MMult( m, n, k, a, lda, b, ldb, cref, ldc );

// repeat the experiment NREPEATS times (from parameters.h)
    /* Time the "optimized" implementation */
    for ( rep=0; rep<NREPEATS; rep++ ){
      copy_matrix( m, n, cold, ldc, c, ldc );

// we measure the execution time
      /* Time your implementation */
      dtime = dclock();

// execute the optimized M*M
      MY_MMult( m, n, k, a, lda, b, ldb, c, ldc );

      dtime = dclock() - dtime;

// we store the best execution time
      if ( rep==0 )
        dtime_best = dtime;
      else
        dtime_best = ( dtime < dtime_best ? dtime :
dtime_best );
    }

// we make sure that both matrices are identical
// (resulting from M*M before and after the optimization)
    diff = compare_matrices( m, n, c, ldc, cref, ldc );

// We compare theoretical gflops to measured gflops
    printf( "%d %le %le \n", p, gflops / dtime_best, diff );
    fflush( stdout );

    free( a );
    free( b );
    free( c );
    free( cold );
    free( cref );
  }
  printf( "];\n" );
  exit( 0 );
}
```

Setup processors parameters

```
% Indicate the number of floating point operations
% that can be executed per clock cycle

nflops_per_cycle = 4;

% Indicate the number of processors being used
% (in case you are using a multicore or SMP)

nprocessors = 1;

% Indicate the clock speed of the processor.
% On a Linux machine this info can be found in the file
% /proc/cpuinfo

% Note: some processors have a "turbo boost" mode,
% which increases the peak clock rate...

GHz_of_processor = 2.6;
```

```
max_gflops = nflops_per_cycle * nprocessors *GHz_of_processor;
max_gflops = 4 * 1 * 2.6 = 10.4
```

Execute

- `make run`
  This will compile, link, and execute the test driver, linking to the implementation in `MMult0.c`. The performance data is saved in file `output0.m`.
- `more output0.m`
  This will display the contents of the output file `output_MMult0.m`.

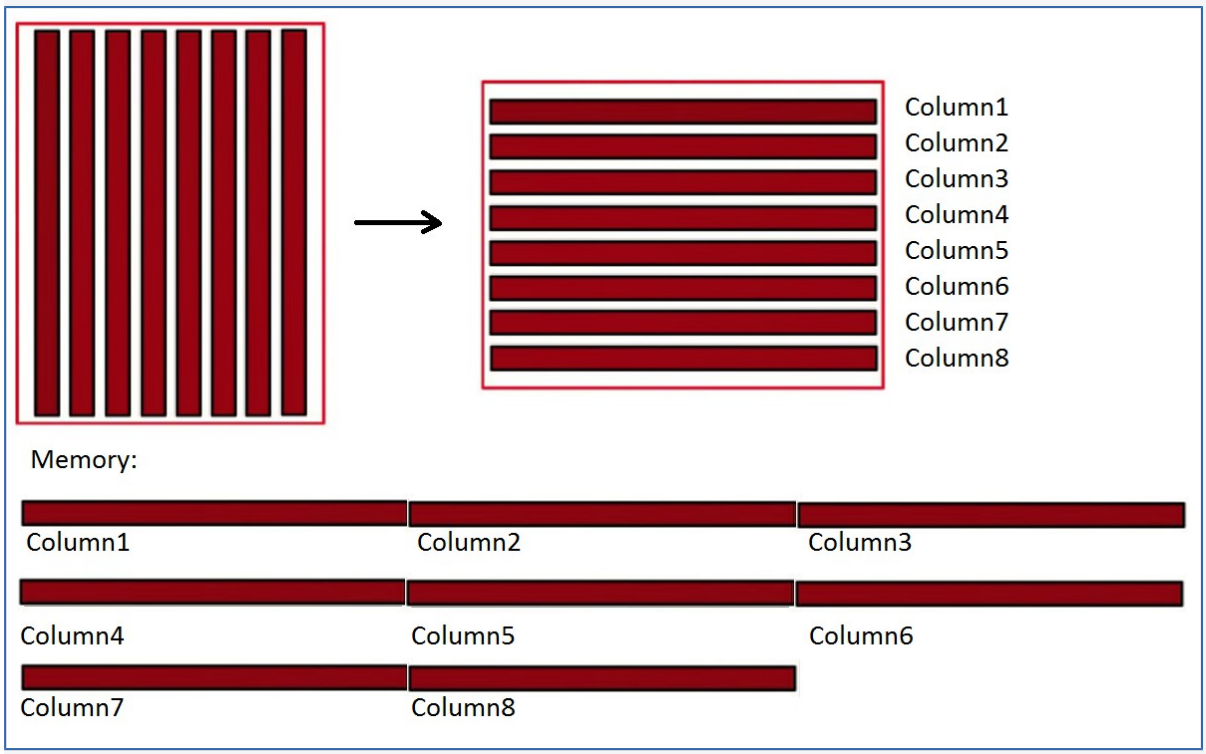
```
version = 'MMult0';
MY_MMult = [
%Matrix size, Gflops, difference between model and optimized code results
40 1.163636e+00 0.000000e+00
80 8.827586e-01 0.000000e+00
120 1.289071e+00 0.000000e+00
160 1.200469e+00 0.000000e+00
[ lines deleted ]
800 2.115609e-01 0.000000e+00
];
```

- `octave`

```
octave:1> PlotAll        % this will create the plot
```

Step 1. Naive Matrix*Matrix multiplication

```
/* Create macros so that the matrices are stored in column-
major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

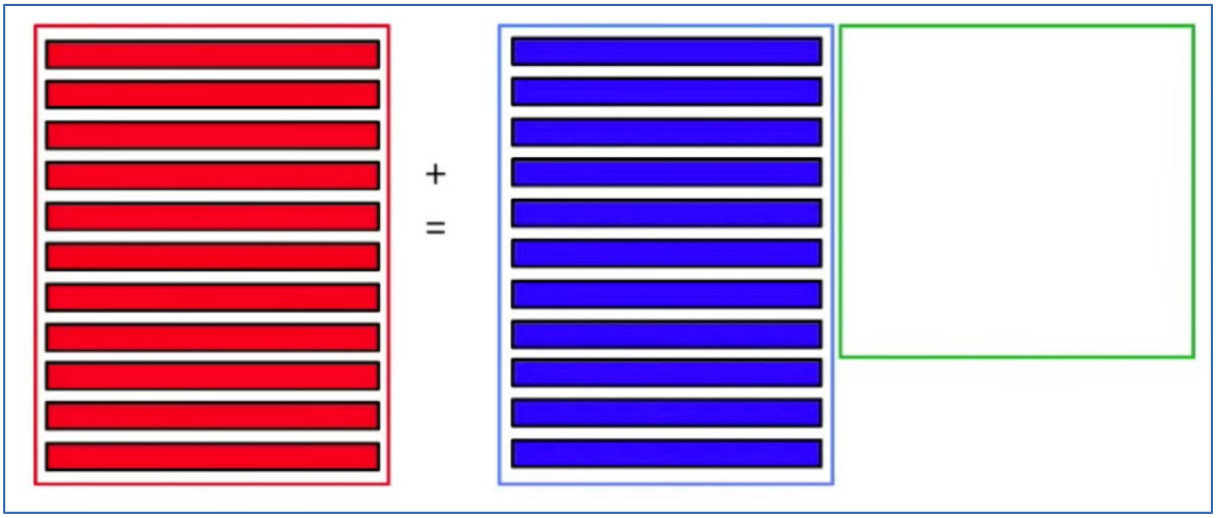



```
/* Routine for computing C = A * B + C */

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j, p;

  for ( i=0; i<m; i++ ){      /* Loop over the rows of C */
    for ( j=0; j<n; j++ ){    /* Loop over the columns of C */
      for ( p=0; p<k; p++ ){ /* Update C( i,j ) with the inner
                                product of the ith row of A and
                                the jth column of B */
        C( i,j ) = C( i,j ) +  A( i,p ) * B( p,j );
      }
    }
  }
```

```
}
```

```
/* Routine for computing C = A * B + C */

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j, p;

  for ( i=0; i<m; i++ ){      /* Loop over the rows of C */
    for ( j=0; j<n; j++ ){    /* Loop over the columns of C */
      for ( p=0; p<k; p++ ){ /* Update C( i,j ) with the inner
                                product of the ith row of A and
                                the jth column of B */
        C( i,j ) = C( i,j ) +  A( i,p ) * B( p,j );
      }
    }
  }
}
```

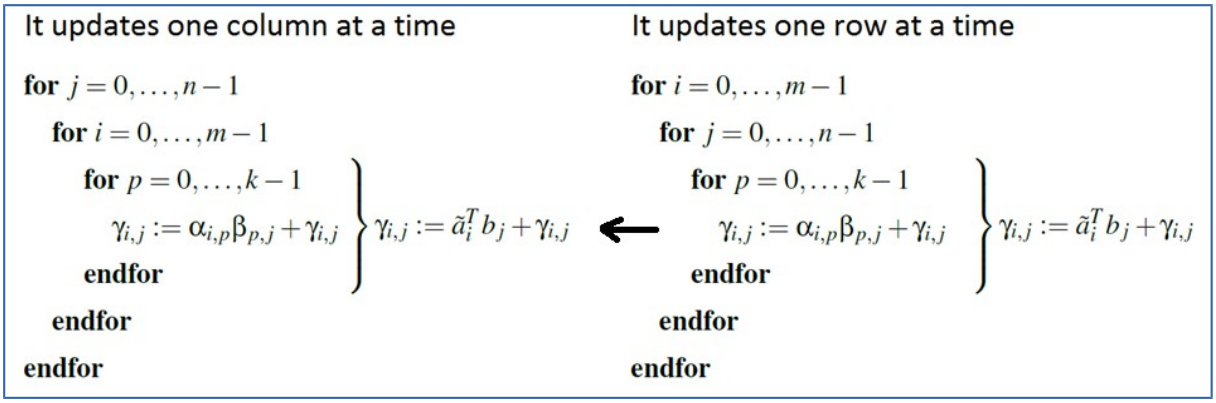
It updates one row at a time

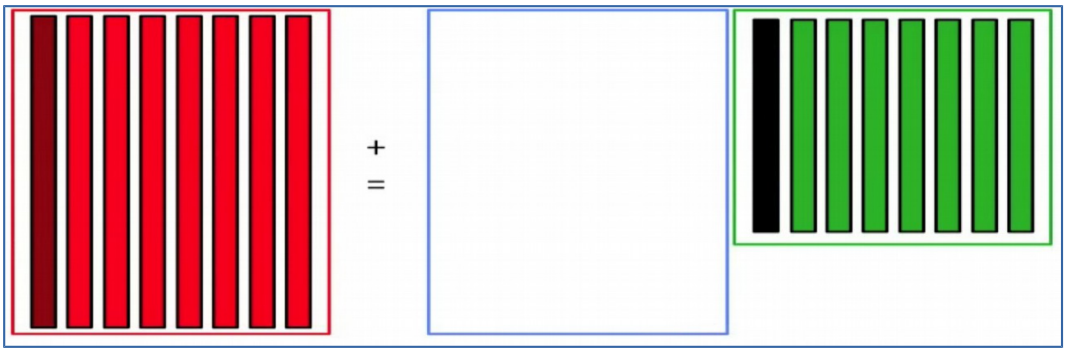


Step 2

```
/* Create macros so that the matrices are stored in column-
major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot( int, double *, int, double *, double * );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=1 ){    /* Loop over the columns of C */
    for ( i=0; i<m; i+=1 ){  /* Loop over the rows of C */
      /* Update the C( i,j ) with the inner product
         of the ith row of A and the jth column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j ), &C( i,j ) );
    }
  }
}
```

First, we went from one column at a time into one row at a time





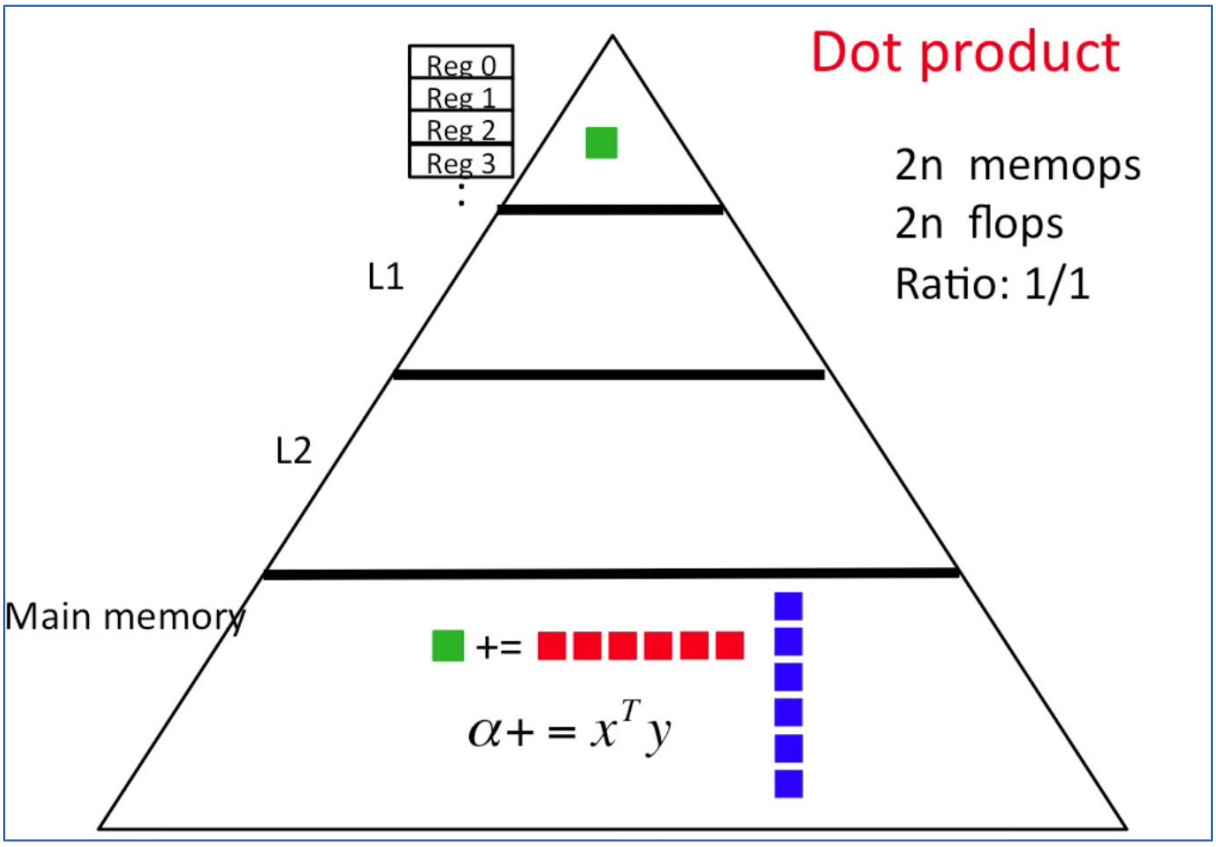
```
void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=1 ){    /* Loop over the columns of C */
    for ( i=0; i<m; i+=1 ){  /* Loop over the rows of C */
       /* Update the C( i,j ) with the inner product
          of the ith row of A and the jth column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j ), &C( i,j ) );
    }
  }
}
```

Second, we have replaced the inner loop by the `AddDot` routine call (we have introduced the dot product)

```
      AddDot( k, &A( i,0 ), lda, &B( 0,j ), &C( i,j ) );
k - size of the row at A, size of the column at B

&A(i,0) - ith row of A (but A is stored column-wise,
so the lda is used to access the row entries

&B(0,j) -jth column of B (but B is stored column-wise,
so direct access is possible

&C(i,j) a single entry of C where the result of dot product is
stored
```

```
/* Create macro to let X( i ) equal the ith element of x */
// The vector is really a row of A, stored column-wise, so the
row is read with the increment incx
#define X(i) x[ (i)*incx ]

void AddDot
( int k, double *x, int incx,  double *y, double *gamma )
```

Dot product of row of x and column of y




```
/* Create macro to let X( i ) equal the ith element of x */
// The vector is really a row of A, read with the increment
incx
#define X(i) x[ (i)*incx ]

void AddDot
( int k, double *x, int incx,  double *y, double *gamma )
{
 /* compute gamma := x' * y + gamma
    with vectors x and y of length n.
    Here x starts at location x with increment (stride) incx
    and y starts at location y and has (implicit) stride of 1.
 */

  int p;
  for ( p=0; p<k; p++ ){
    *gamma += X( p ) * y[ p ];
  }
}
```

Step 3.

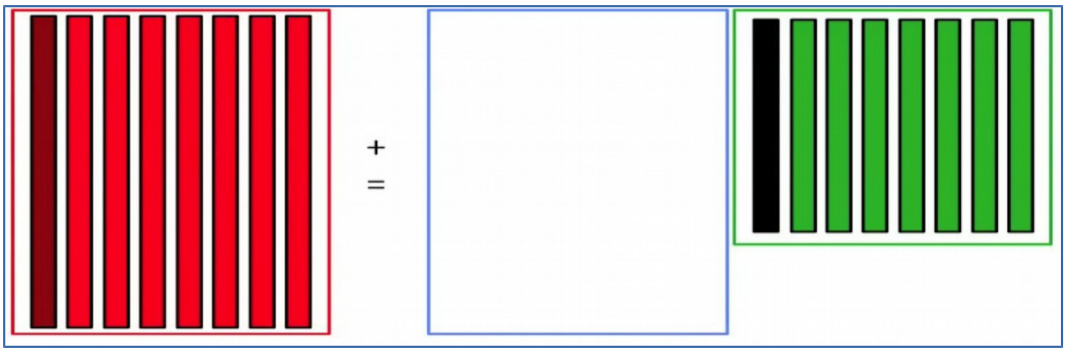
From

```
void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=1 ){    /* Loop over the columns of C */
    for ( i=0; i<m; i+=1 ){  /* Loop over the rows of C */
      /* Update the C( i,j ) with the inner product
         of the ith row of A and the jth column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j ), &C( i,j ) );
    }
  }
}
```

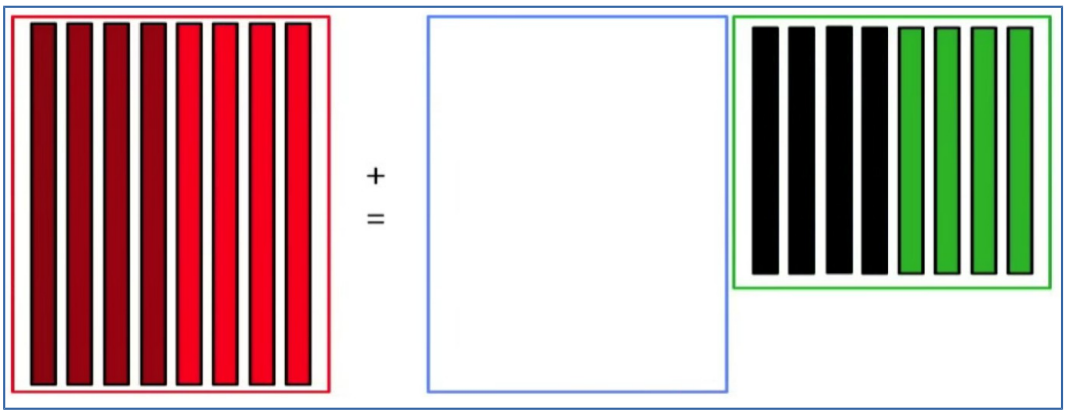
To

```
void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;
  /* Loop over the columns of C, unrolled by 4 */
  for ( j=0; j<n; j+=4 ){
    /* Loop over the rows of C */
    for ( i=0; i<m; i+=1 ){
      /* Update the C( i,j ) with the inner product
         of the ith row of A and the jth column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j ), &C( i,j ) );

      /* Update the C( i,j+1 ) with the inner product
         of the ith row of A and the (j+1)th column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j+1 ), &C( i,j+1 ) );

      /* Update the C( i,j+2 ) with the inner product
         of the ith row of A and the (j+2)th column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j+2 ), &C( i,j+2 ) );

      /* Update the C( i,j+3 ) with the inner product
         of the ith row of A and the (j+1)th column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j+3 ), &C( i,j+3 ) );
    }
  }
}
```

Instead of processing one column at a time

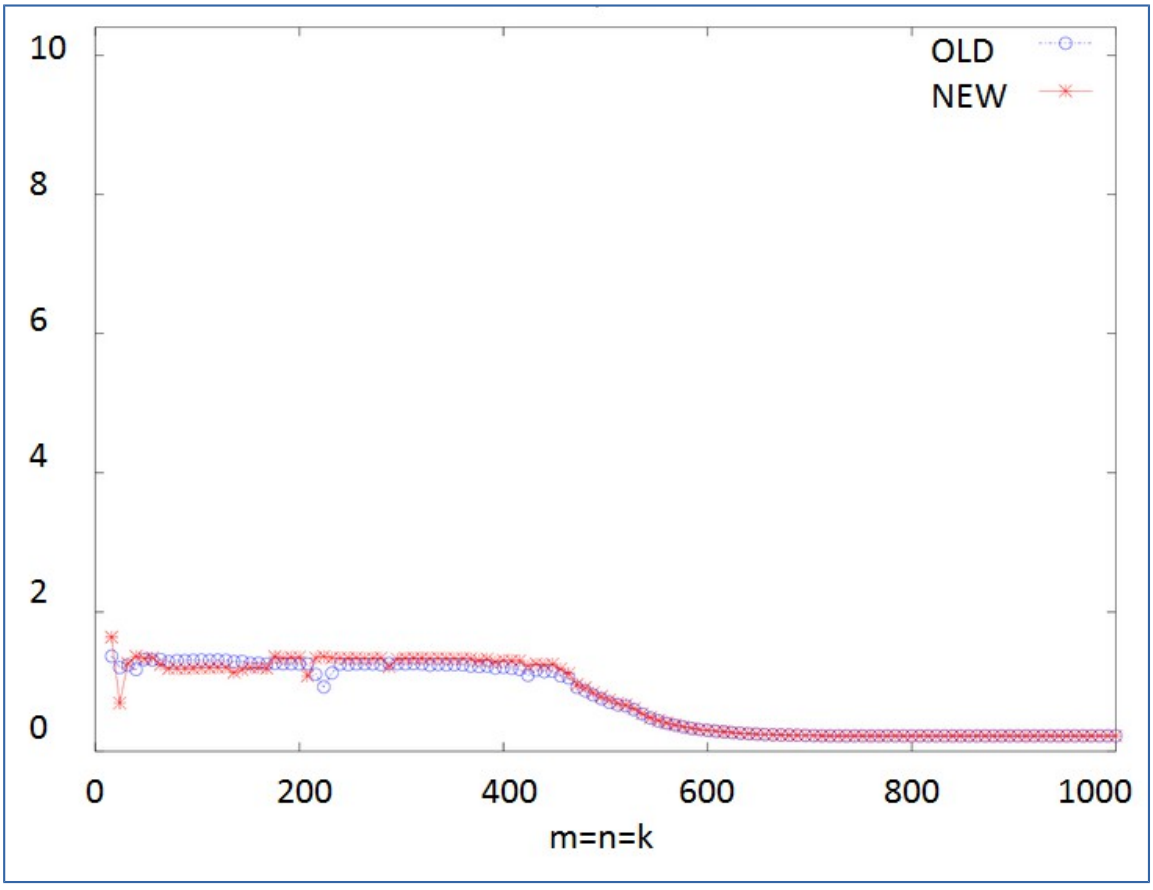


we process now 4 columns at a time



Result: (doesn't work YET)

Gflops/s



Step 4.

From

```
void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;
  /* Loop over the columns of C, unrolled by 4 */
  for ( j=0; j<n; j+=4 ){
    /* Loop over the rows of C */
    for ( i=0; i<m; i+=1 ){
      /* Update the C( i,j ) with the inner product
         of the ith row of A and the jth column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j ), &C( i,j ) );

      /* Update the C( i,j+1 ) with the inner product
         of the ith row of A and the (j+1)th column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j+1 ), &C( i,j+1 ) );

      /* Update the C( i,j+2 ) with the inner product
         of the ith row of A and the (j+2)th column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j+2 ), &C( i,j+2 ) );

      /* Update the C( i,j+3 ) with the inner product
         of the ith row of A and the (j+1)th column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j+3 ), &C( i,j+3 ) );
    }
  }
}
```

To

```
void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;
  /* Loop over the columns of C, unrolled by 4 */
  for ( j=0; j<n; j+=4 ){
    /* Loop over the rows of C */
    for ( i=0; i<m; i+=1 ){
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ),and C( i,j+3)
         in one routine (four inner products) */
      AddDot1x4
        ( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}
```

Call one routine `AddDot1x4` instead of four routine `AddDot` from the main loops

```
AddDot1x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc);
k - size of the row at A, size of the column at B

&A(i,0) - ith row of A (but A is stored column-wise,
so the lda is used to access the row entries

&B(0,j) -jth column of B (but B is stored column-wise,
so direct access is possible

&C(i,j) a single entry of C where the result of dot product is
stored
```

```
void AddDot1x4( int k, double *a, int lda,
                       double *b, int ldb,
                       double *c, int ldc )
{
  /* So, this routine computes four elements of C:

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).

  Notice that this routine is called with c = C( i, j ) in
  the previous routine, so these are actually the elements

           C( i, j ), C( i, j+1 ), C( i, j+2 ), C( i, j+3 )

     in the original matrix C */

  AddDot( k, &A( 0, 0 ), lda, &B( 0, 0 ), &C( 0, 0 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 1 ), &C( 0, 1 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 2 ), &C( 0, 2 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 3 ), &C( 0, 3 ) );
}
```

But routine `AddDot1x4` calles four times routine `AddDot`
but now we can inline it

Step 5

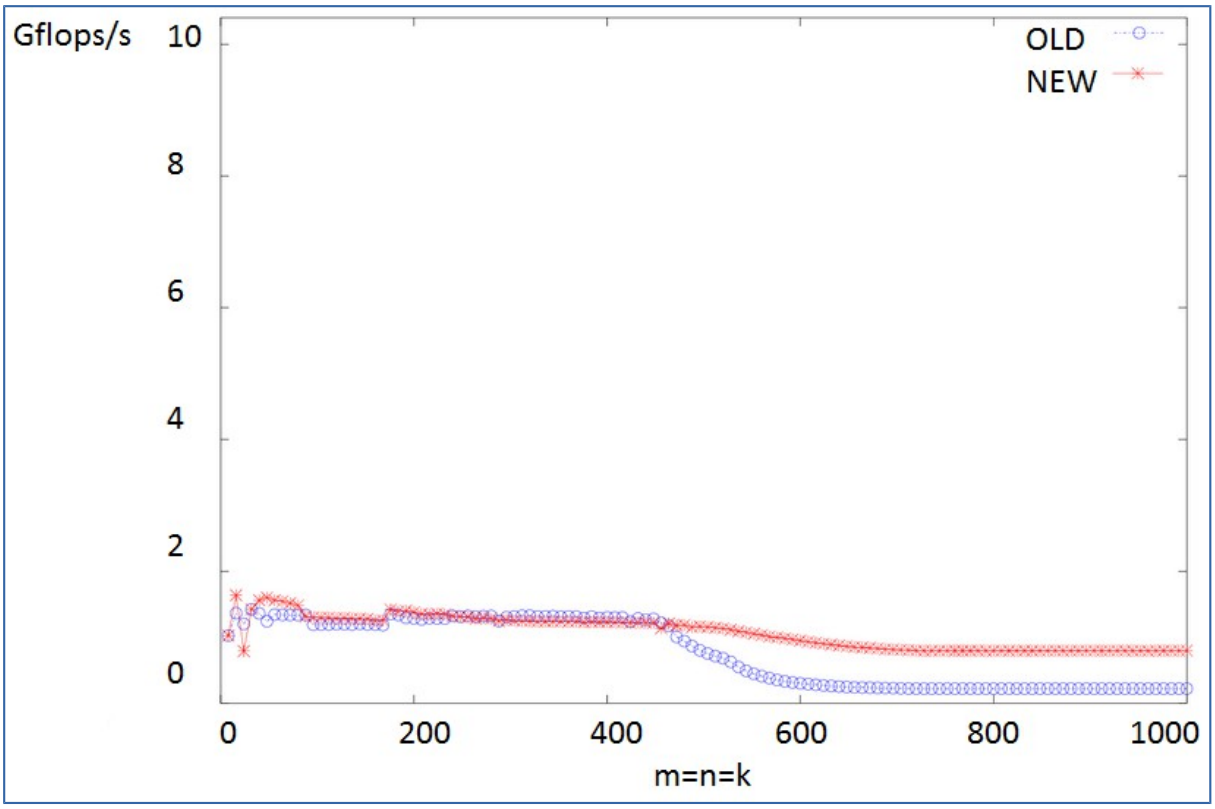
We inline the routine `AddDot`

```
void AddDot1x4( int k, double *a, int lda,
                       double *b, int ldb,
                       double *c, int ldc )
{
  /* So, this routine computes four elements of C:

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).

  Notice that this routine is called with c = C( i, j ) in
  the previous routine, so these are actually the elements

           C( i, j ), C( i, j+1 ), C( i, j+2 ), C( i, j+3 )

     in the original matrix C

     In this version, we "inline" AddDot */

  int p;

  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 0 ), &C( 0, 0 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 0 ) += A( 0, p ) * B( p, 0 );
  }

  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 1 ), &C( 0, 1 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 1 ) += A( 0, p ) * B( p, 1 );
  }

  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 2 ), &C( 0, 2 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 2 ) += A( 0, p ) * B( p, 2 );
  }

  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 3 ), &C( 0, 3 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 3 ) += A( 0, p ) * B( p, 3 );
  }
}
```

Step 6

We make one loop out of four

```
void AddDot1x4( int k, double *a, int lda,
                       double *b, int ldb,
                       double *c, int ldc )
{
  /* So, this routine computes four elements of C:

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).

  Notice that this routine is called with c = C( i, j ) in
  the previous routine, so these are actually the elements

           C( i, j ), C( i, j+1 ), C( i, j+2 ), C( i, j+3 )

     in the original matrix C

     In this version, we merge the four loops,
     computing four inner products simultaneously. */

  int p;

  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 0 ), &C( 0, 0 ) );
  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 1 ), &C( 0, 1 ) );
  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 2 ), &C( 0, 2 ) );
  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 3 ), &C( 0, 3 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 0 ) += A( 0, p ) * B( p, 0 );
    C( 0, 1 ) += A( 0, p ) * B( p, 1 );
    C( 0, 2 ) += A( 0, p ) * B( p, 2 );
    C( 0, 3 ) += A( 0, p ) * B( p, 3 );
  }
}
```



Now we start seeing a performance benefit.

The reason is that the four loops have been fused and therefore the four inner products are now being performed simultaneously. This has the following benefits:

- The index `p` needs only be updated once every eight floating point operations.
- Element `A( 0, p )` needs only be brought in from memory once instead of four times. (This only becomes a benefit when the matrices no longer fit in the L2 cache.)

Step 7

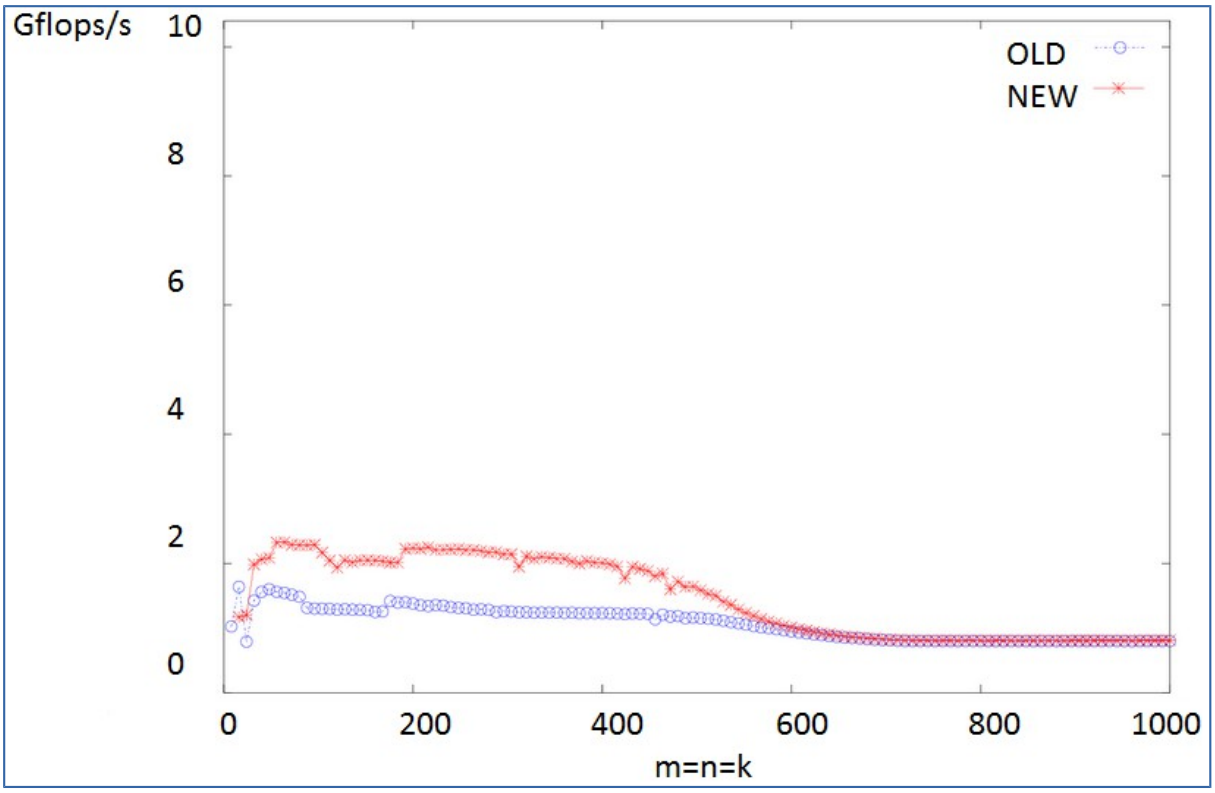
Use register variables

From

```
void AddDot1x4( int k, double *a, int lda,  double *b, int ldb,
double *c, int ldc )
{
  int p;

  for ( p=0; p<k; p++ ){
    C( 0, 0 ) += A( 0, p ) * B( p, 0 );
    C( 0, 1 ) += A( 0, p ) * B( p, 1 );
    C( 0, 2 ) += A( 0, p ) * B( p, 2 );
    C( 0, 3 ) += A( 0, p ) * B( p, 3 );
  }
}
```

to

```
void AddDot1x4( int k, double *a, int lda,  double *b, int ldb,
double *c, int ldc )
{
  int p;
    /* hold contributions to in register
       C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ) */
       c_00_reg,   c_01_reg,   c_02_reg,   c_03_reg,
  register double c_00_reg,   c_01_reg,   c_02_reg,  c_03_reg;

    /* holds A( 0, p ) */
  register double a_0p_reg;

  c_00_reg = 0.0;
  c_01_reg = 0.0;
  c_02_reg = 0.0;
  c_03_reg = 0.0;
   for ( p=0; p<k; p++ ){
    a_0p_reg = A( 0, p );
    c_00_reg += a_0p_reg * B( p, 0 );
    c_01_reg += a_0p_reg * B( p, 1 );
    c_02_reg += a_0p_reg * B( p, 2 );
    c_03_reg += a_0p_reg * B( p, 3 );
  }

  C( 0, 0 ) += c_00_reg;
  C( 0, 1 ) += c_01_reg;
  C( 0, 2 ) += c_02_reg;
  C( 0, 3 ) += c_03_reg;
}
```



Now we start seeing a performance benefit. We accumulate the updates to the current 1x4 row of C in registers and we place the element A(p, 0) in a register, to reduce traffic between cache and registers.

Step 8

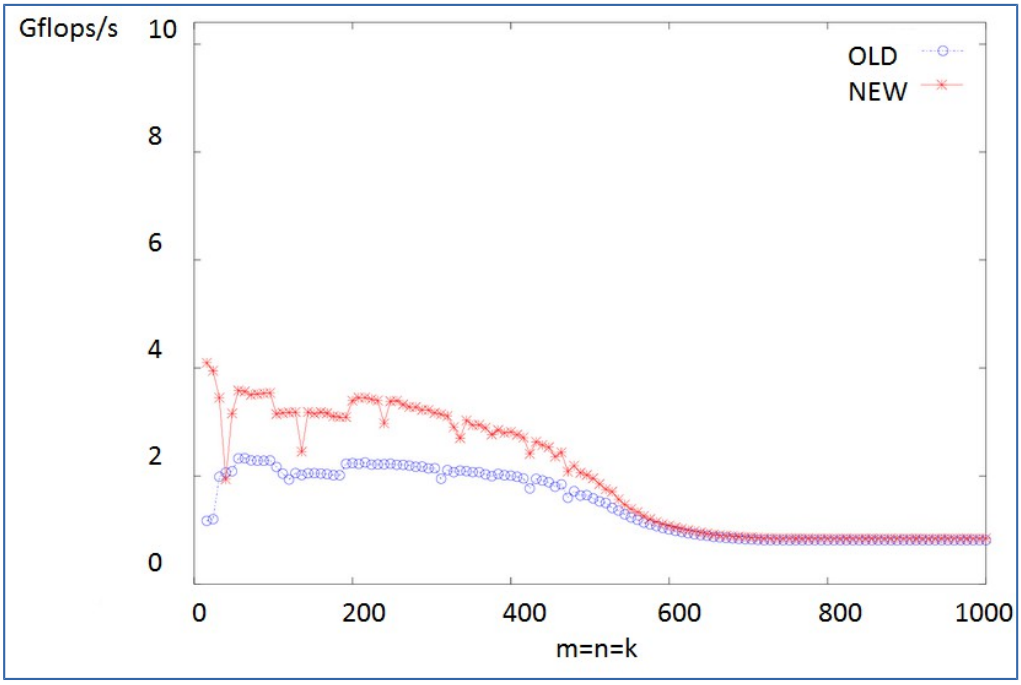
Access matrices through pointers

From

```
c_00_reg = 0.0;
c_01_reg = 0.0;
c_02_reg = 0.0;
c_03_reg = 0.0;

for ( p=0; p<k; p++ ){
  a_0p_reg = A( 0, p );

  c_00_reg += a_0p_reg * B( p, 0 );
  c_01_reg += a_0p_reg * B( p, 1 );
  c_02_reg += a_0p_reg * B( p, 2 );
  c_03_reg += a_0p_reg * B( p, 3 );
}

C( 0, 0 ) += c_00_reg;
C( 0, 1 ) += c_01_reg;
C( 0, 2 ) += c_02_reg;
C( 0, 3 ) += c_03_reg;
```

to

```
  bp0_pntr = &B( 0, 0 );
  bp1_pntr = &B( 0, 1 );
  bp2_pntr = &B( 0, 2 );
  bp3_pntr = &B( 0, 3 );

  c_00_reg = 0.0;
  c_01_reg = 0.0;
  c_02_reg = 0.0;
  c_03_reg = 0.0;

  for ( p=0; p<k; p++ ){
    a_0p_reg = A( 0, p );

    c_00_reg += a_0p_reg * *bp0_pntr++;
    c_01_reg += a_0p_reg * *bp1_pntr++;
    c_02_reg += a_0p_reg * *bp2_pntr++;
    c_03_reg += a_0p_reg * *bp3_pntr++;
  }

  C( 0, 0 ) += c_00_reg;
  C( 0, 1 ) += c_01_reg;
  C( 0, 2 ) += c_02_reg;
  C( 0, 3 ) += c_03_reg;
```



We now use four pointers, `bp0_pntr`, `bp1_pntr`, `bp2_pntr`,and `bp3_pntr`, to access the elements `B( p, 0 )`,`B( p, 1 )`,`B( p, 2 )`,`B( p, 3 )`. This reduces indexing overhead.

Step 9

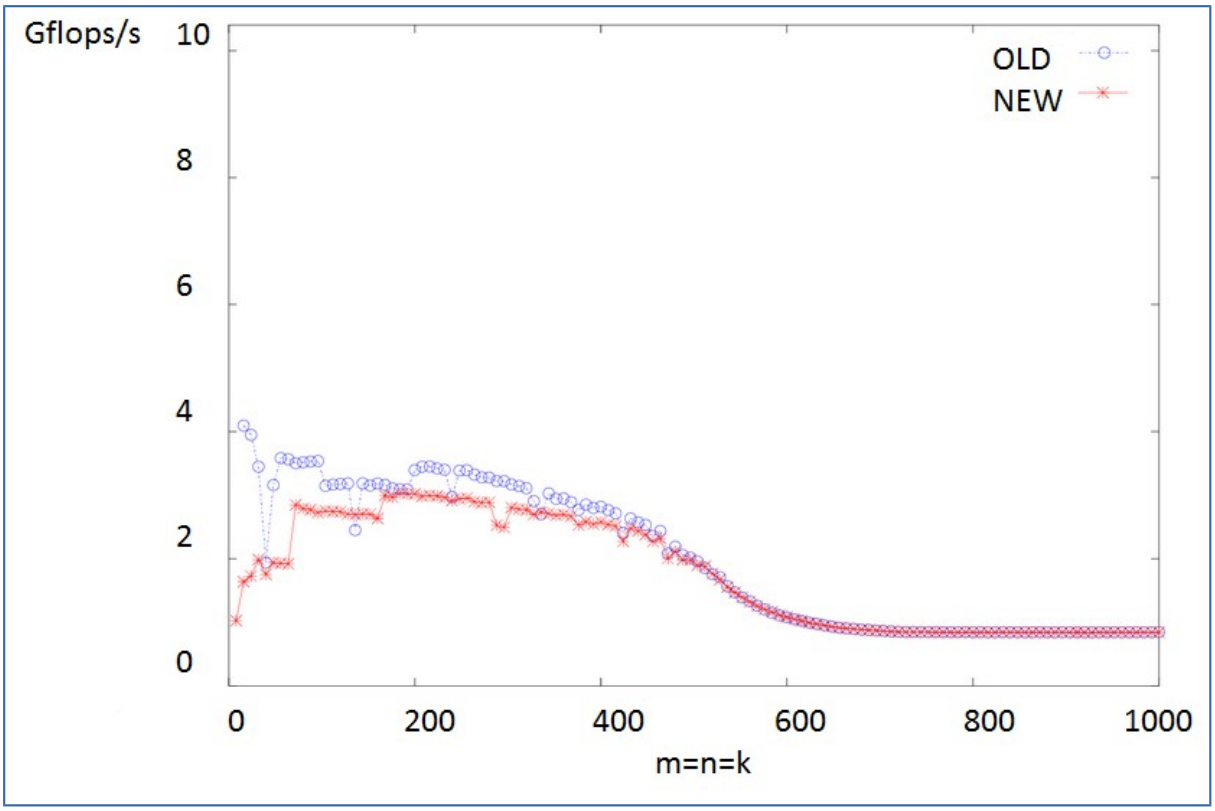
Unroll the loop by four (arbitrary selected value)

From

```
for ( p=0; p<k; p++ ){
    a_0p_reg = A( 0, p );

    c_00_reg += a_0p_reg * *bp0_pntr++;
    c_01_reg += a_0p_reg * *bp1_pntr++;
    c_02_reg += a_0p_reg * *bp2_pntr++;
    c_03_reg += a_0p_reg * *bp3_pntr++;
  }
```

to

```
for ( p=0; p<k; p+=4 ){
    a_0p_reg = A( 0, p );

    c_00_reg += a_0p_reg * *bp0_pntr++;
    c_01_reg += a_0p_reg * *bp1_pntr++;
    c_02_reg += a_0p_reg * *bp2_pntr++;
    c_03_reg += a_0p_reg * *bp3_pntr++;

    a_0p_reg = A( 0, p+1 );

    c_00_reg += a_0p_reg * *bp0_pntr++;
    c_01_reg += a_0p_reg * *bp1_pntr++;
    c_02_reg += a_0p_reg * *bp2_pntr++;
    c_03_reg += a_0p_reg * *bp3_pntr++;

    a_0p_reg = A( 0, p+2 );

    c_00_reg += a_0p_reg * *bp0_pntr++;
    c_01_reg += a_0p_reg * *bp1_pntr++;
    c_02_reg += a_0p_reg * *bp2_pntr++;
    c_03_reg += a_0p_reg * *bp3_pntr++;

    a_0p_reg = A( 0, p+3 );

    c_00_reg += a_0p_reg * *bp0_pntr++;
    c_01_reg += a_0p_reg * *bp1_pntr++;
    c_02_reg += a_0p_reg * *bp2_pntr++;
    c_03_reg += a_0p_reg * *bp3_pntr++;
  }
```



We now unrolled the loop by four. Interestingly enough, this decreases performance slightly. What this probably means is that by adding the optimization, we confused the compiler, which therefore could not do an optimization that it did before.

Step 10

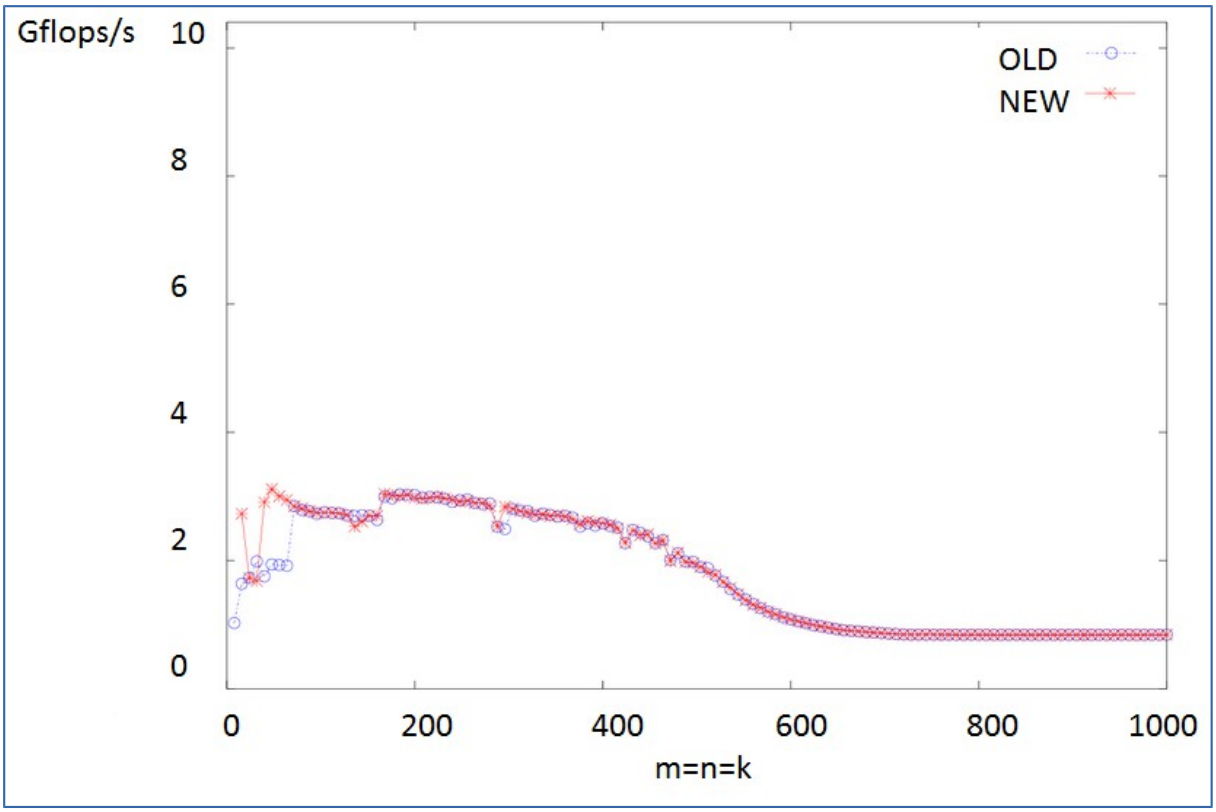
Use indirect adressing to reduce the number of times the pointers need to be updated

From

```
  bp0_pntr = &B( 0, 0 );
  bp1_pntr = &B( 0, 1 );
  bp2_pntr = &B( 0, 2 );
  bp3_pntr = &B( 0, 3 );

  c_00_reg = 0.0;
  c_01_reg = 0.0;
  c_02_reg = 0.0;
  c_03_reg = 0.0;
  for ( p=0; p<k; p+=4 ){
    a_0p_reg = A( 0, p );

    c_00_reg += a_0p_reg * *bp0_pntr++;
    c_01_reg += a_0p_reg * *bp1_pntr++;
    c_02_reg += a_0p_reg * *bp2_pntr++;
    c_03_reg += a_0p_reg * *bp3_pntr++;

    a_0p_reg = A( 0, p+1 );

    c_00_reg += a_0p_reg * *bp0_pntr++;
    c_01_reg += a_0p_reg * *bp1_pntr++;
    c_02_reg += a_0p_reg * *bp2_pntr++;
    c_03_reg += a_0p_reg * *bp3_pntr++;

    a_0p_reg = A( 0, p+2 );

    c_00_reg += a_0p_reg * *bp0_pntr++;
    c_01_reg += a_0p_reg * *bp1_pntr++;
    c_02_reg += a_0p_reg * *bp2_pntr++;
    c_03_reg += a_0p_reg * *bp3_pntr++;

    a_0p_reg = A( 0, p+3 );

    c_00_reg += a_0p_reg * *bp0_pntr++;
    c_01_reg += a_0p_reg * *bp1_pntr++;
    c_02_reg += a_0p_reg * *bp2_pntr++;
    c_03_reg += a_0p_reg * *bp3_pntr++;
  }

  C( 0, 0 ) += c_00_reg;
  C( 0, 1 ) += c_01_reg;
  C( 0, 2 ) += c_02_reg;
  C( 0, 3 ) += c_03_reg;
}
```

to

```
  bp0_pntr = &B( 0, 0 );
  bp1_pntr = &B( 0, 1 );
  bp2_pntr = &B( 0, 2 );
  bp3_pntr = &B( 0, 3 );

  c_00_reg = 0.0;
  c_01_reg = 0.0;
  c_02_reg = 0.0;
  c_03_reg = 0.0;
  for ( p=0; p<k; p+=4 ){
    a_0p_reg = A( 0, p );

    c_00_reg += a_0p_reg * *bp0_pntr;
    c_01_reg += a_0p_reg * *bp1_pntr;
    c_02_reg += a_0p_reg * *bp2_pntr;
    c_03_reg += a_0p_reg * *bp3_pntr;

    a_0p_reg = A( 0, p+1 );

    c_00_reg += a_0p_reg * *(bp0_pntr+1);
    c_01_reg += a_0p_reg * *(bp1_pntr+1);
    c_02_reg += a_0p_reg * *(bp2_pntr+1);
    c_03_reg += a_0p_reg * *(bp3_pntr+1);

    a_0p_reg = A( 0, p+2 );

    c_00_reg += a_0p_reg * *(bp0_pntr+2);
    c_01_reg += a_0p_reg * *(bp1_pntr+2);
    c_02_reg += a_0p_reg * *(bp2_pntr+2);
    c_03_reg += a_0p_reg * *(bp3_pntr+2);

    a_0p_reg = A( 0, p+3 );

    c_00_reg += a_0p_reg * *(bp0_pntr+3);
    c_01_reg += a_0p_reg * *(bp1_pntr+3);
    c_02_reg += a_0p_reg * *(bp2_pntr+3);
    c_03_reg += a_0p_reg * *(bp3_pntr+3);

    bp0_pntr+=4;
    bp1_pntr+=4;
    bp2_pntr+=4;
    bp3_pntr+=4;
  }
  C( 0, 0 ) += c_00_reg;
  C( 0, 1 ) += c_01_reg;
  C( 0, 2 ) += c_02_reg;
  C( 0, 3 ) += c_03_reg;
}
```



We now use something called 'indirect addressing'. Notice, for example, the line

```
    c_00_reg += a_0p_reg * *(bp0_pntr+1);
```

Here

- `a0p_reg` holds the element `A( 0, p+1 )` (yes, this is a bit confusing. A better name for the variable would be good...)
- We want to `bp0_pntr' points to element` B(p, 0).
  Hence bp0_pntr+1' addresses the element `B( p+1, 0 )`. There is a special machine instruction to then access the element at `bp0_pntr+1' that does not require the pointer to be updated.
- As a result, the pointers that address the elements in the columns of B only need to be updated once every fourth iteration of the loop.

Interestingly, it appears that the compiler did this optimization automatically, and hence we see no performance improvement...

What happens if we switch into processing 4x4 blocks of the matrix?

We now compute a 4 x 4 block of matrix at a time
in order to use **vector instructions** and **vector registers** effectively.

We have used compilation flag:
-msse3 (generates floating point operations for specific unit x86 in this case)

SSE3 instructions --> multiple operations per clock cycle
two multiplies and two addes per clock
4 flops per clock
It is necessary to use 'vector registers'
There are 16 vector registers (each can hold 2 double precision number)
2*16=32 double precisions, for which we can perform 4 operations per clock (two multiplications and two additions)
4x4 block = 16 double precisions (we can put 16 doubles in these vector registers)

We go back to version with 4 blocked AddDot calls

```
void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;
  /* Loop over the columns of C, unrolled by 4 */
  for ( j=0; j<n; j+=4 ){
    /* Loop over the rows of C */
    for ( i=0; i<m; i+=1 ){
      /* Update the C( i,j ) with the inner product
         of the ith row of A and the jth column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j ), &C( i,j ) );

      /* Update the C( i,j+1 ) with the inner product
         of the ith row of A and the (j+1)th column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j+1 ), &C( i,j+1 ) );

      /* Update the C( i,j+2 ) with the inner product
         of the ith row of A and the (j+2)th column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j+2 ), &C( i,j+2 ) );

      /* Update the C( i,j+3 ) with the inner product
         of the ith row of A and the (j+1)th column of B */
      AddDot( k, &A( i,0 ), lda, &B( 0,j+3 ), &C( i,j+3 ) );
    }
```

```
  }
}
```

Add +=4 in the second loop (now we group to perform operations on 4x4 blocks)

```
void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;
  /* Loop over the columns of C, unrolled by 4 */
  for ( j=0; j<n; j+=4 ){
    /* Loop over the rows of C */
    for ( i=0; i<m; i+=4 ){
    /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 )
       in one routine (four inner products) */
    AddDot4x4(k, &A(i,0 ), lda, &B(0,j ), ldb, &C(i,j ), ldc);
    }
  }
}
```

We need to implement `AddDot4x4`

```
void AddDot4x4( int k, double *a, int lda,
                       double *b, int ldb,
                       double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A
           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

   Notice that this routine is called with c = C( i, j )
   in the previous routine, so these are actually the elements

   C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
   C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
   C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
   C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

     in the original matrix C */

  /* First row */
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 0 ), &C( 0, 0 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 1 ), &C( 0, 1 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 2 ), &C( 0, 2 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 3 ), &C( 0, 3 ) );
```

```
  /* Second row */
  AddDot( k, &A( 1, 0 ), lda, &B( 0, 0 ), &C( 1, 0 ) );
  AddDot( k, &A( 1, 0 ), lda, &B( 0, 1 ), &C( 1, 1 ) );
  AddDot( k, &A( 1, 0 ), lda, &B( 0, 2 ), &C( 1, 2 ) );
  AddDot( k, &A( 1, 0 ), lda, &B( 0, 3 ), &C( 1, 3 ) );

  /* Third row */
  AddDot( k, &A( 2, 0 ), lda, &B( 0, 0 ), &C( 2, 0 ) );
  AddDot( k, &A( 2, 0 ), lda, &B( 0, 1 ), &C( 2, 1 ) );
  AddDot( k, &A( 2, 0 ), lda, &B( 0, 2 ), &C( 2, 2 ) );
  AddDot( k, &A( 2, 0 ), lda, &B( 0, 3 ), &C( 2, 3 ) );

  /* Four row */
  AddDot( k, &A( 3, 0 ), lda, &B( 0, 0 ), &C( 3, 0 ) );
  AddDot( k, &A( 3, 0 ), lda, &B( 0, 1 ), &C( 3, 1 ) );
  AddDot( k, &A( 3, 0 ), lda, &B( 0, 2 ), &C( 3, 2 ) );
  AddDot( k, &A( 3, 0 ), lda, &B( 0, 3 ), &C( 3, 3 ) );
}
```

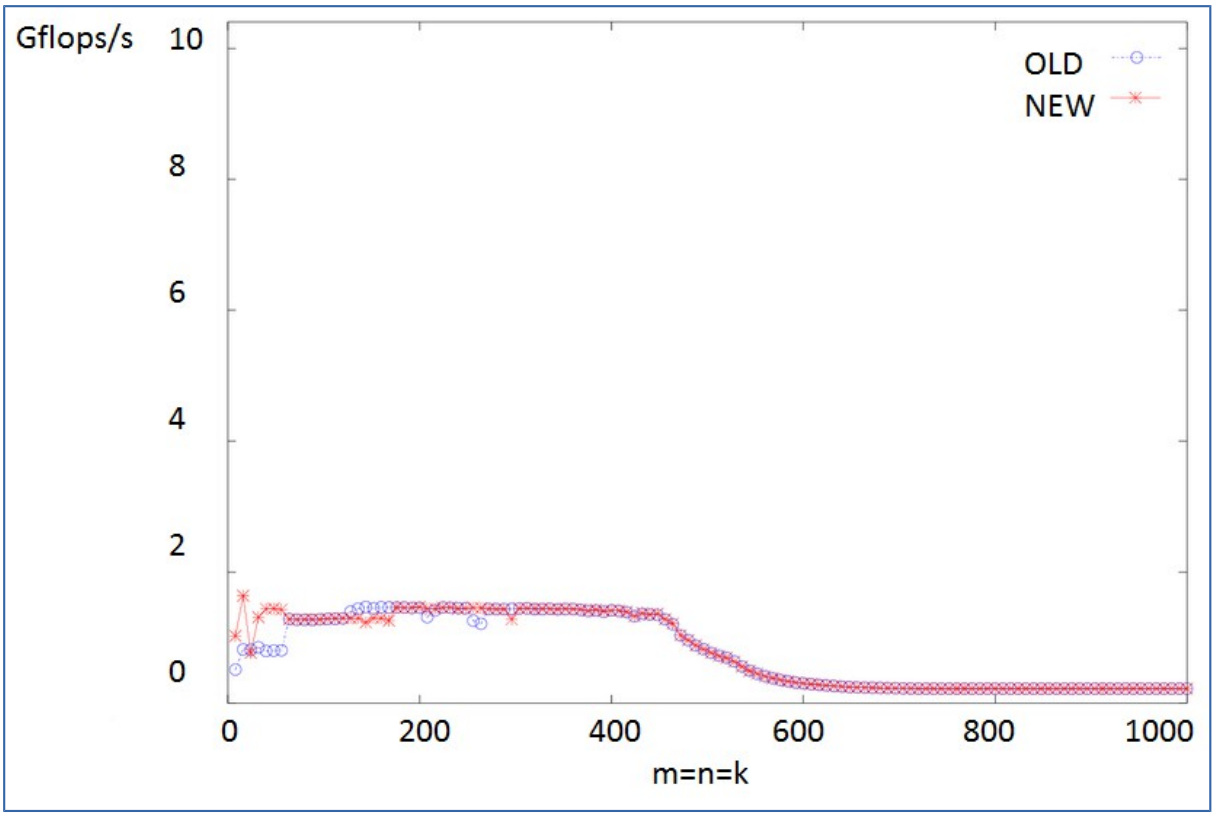
Analogous case for 2x2 blocks:

$$\left(\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right) = \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right)\left(\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right)$$

$$= \left(\begin{array}{c|c} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ \hline A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{array}\right)$$

$$\left(\begin{array}{cc|cc} \gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\ \gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{1,3} \\ \hline \gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{2,3} \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{3,3} \end{array}\right) = \left(\begin{array}{cc|cc} \alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} & \alpha_{0,3} \\ \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} \\ \hline \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} \\ \alpha_{3,0} & \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} \end{array}\right)\left(\begin{array}{cc|cc} \beta_{0,0} & \beta_{0,1} & \beta_{0,2} & \beta_{0,3} \\ \beta_{1,0} & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} \\ \hline \beta_{2,0} & \beta_{2,1} & \beta_{2,2} & \beta_{2,3} \\ \beta_{3,0} & \beta_{3,1} & \beta_{3,2} & \beta_{3,3} \end{array}\right) =$$

$$\left(\begin{array}{c|c} \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} \\ \alpha_{1,0} & \alpha_{1,1} \end{pmatrix}\begin{pmatrix} \beta_{0,0} & \beta_{0,1} \\ \beta_{1,0} & \beta_{1,1} \end{pmatrix} + \begin{pmatrix} \alpha_{0,2} & \alpha_{0,3} \\ \alpha_{1,2} & \alpha_{1,3} \end{pmatrix}\begin{pmatrix} \beta_{2,0} & \beta_{2,1} \\ \beta_{3,0} & \beta_{3,1} \end{pmatrix} & \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} \\ \alpha_{1,0} & \alpha_{1,1} \end{pmatrix}\begin{pmatrix} \beta_{0,2} & \beta_{0,3} \\ \beta_{1,2} & \beta_{1,3} \end{pmatrix} + \begin{pmatrix} \alpha_{0,2} & \alpha_{0,3} \\ \alpha_{1,2} & \alpha_{1,3} \end{pmatrix}\begin{pmatrix} \beta_{2,2} & \beta_{2,3} \\ \beta_{3,2} & \beta_{3,3} \end{pmatrix} \\ \hline \begin{pmatrix} \alpha_{2,0} & \alpha_{2,1} \\ \alpha_{3,0} & \alpha_{3,1} \end{pmatrix}\begin{pmatrix} \beta_{0,0} & \beta_{0,1} \\ \beta_{1,0} & \beta_{1,1} \end{pmatrix} + \begin{pmatrix} \alpha_{2,2} & \alpha_{2,3} \\ \alpha_{3,2} & \alpha_{3,3} \end{pmatrix}\begin{pmatrix} \beta_{2,0} & \beta_{2,1} \\ \beta_{3,0} & \beta_{3,1} \end{pmatrix} & \begin{pmatrix} \alpha_{2,0} & \alpha_{2,1} \\ \alpha_{3,0} & \alpha_{3,1} \end{pmatrix}\begin{pmatrix} \beta_{0,2} & \beta_{0,3} \\ \beta_{1,2} & \beta_{1,3} \end{pmatrix} + \begin{pmatrix} \alpha_{2,2} & \alpha_{2,3} \\ \alpha_{3,2} & \alpha_{3,3} \end{pmatrix}\begin{pmatrix} \beta_{2,2} & \beta_{2,3} \\ \beta_{3,2} & \beta_{3,3} \end{pmatrix} \end{array}\right)$$



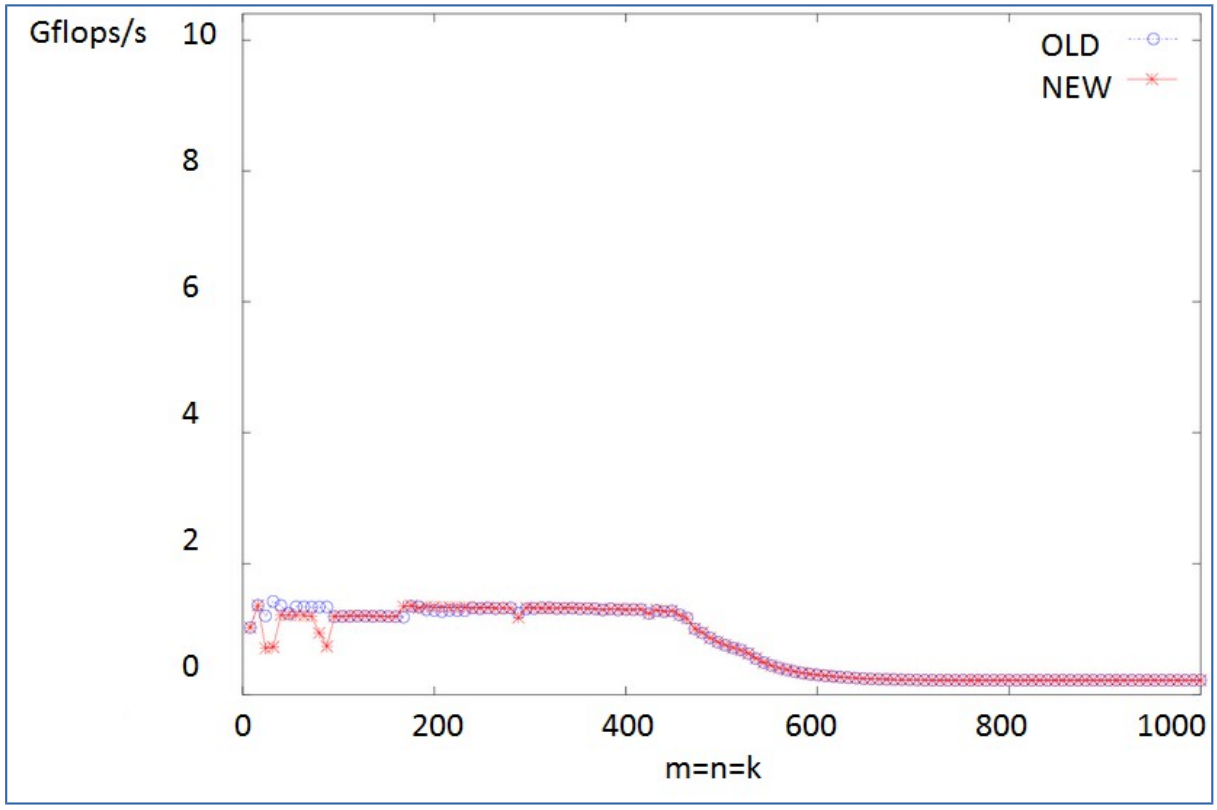
No performance benefit yet

We inline AddDot calls:

```
void AddDot4x4( int k, double *a, int lda,
                       double *b, int ldb,
                       double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

   Notice that this routine is called with c = C( i, j )
   in the previous routine, so these are actually the elements

     C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
     C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
     C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
     C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

   in the original matrix C

   In this version, we "inline" AddDot */

  int p;
```

```
  /* First row */
  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 0 ), &C( 0, 0 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 0 ) += A( 0, p ) * B( p, 0 );
  }
  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 1 ), &C( 0, 1 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 1 ) += A( 0, p ) * B( p, 1 );
  }
  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 2 ), &C( 0, 2 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 2 ) += A( 0, p ) * B( p, 2 );
  }
  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 3 ), &C( 0, 3 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 3 ) += A( 0, p ) * B( p, 3 );
  }

  /* Second row */
  //  AddDot( k, &A( 1, 0 ), lda, &B( 0, 0 ), &C( 1, 0 ) );
  for ( p=0; p<k; p++ ){
    C( 1, 0 ) += A( 1, p ) * B( p, 0 );
  }
  //  AddDot( k, &A( 1, 0 ), lda, &B( 0, 1 ), &C( 1, 1 ) );
  for ( p=0; p<k; p++ ){
    C( 1, 1 ) += A( 1, p ) * B( p, 1 );
  }
  //  AddDot( k, &A( 1, 0 ), lda, &B( 0, 2 ), &C( 1, 2 ) );
  for ( p=0; p<k; p++ ){
    C( 1, 2 ) += A( 1, p ) * B( p, 2 );
  }
  //  AddDot( k, &A( 1, 0 ), lda, &B( 0, 3 ), &C( 1, 3 ) );
  for ( p=0; p<k; p++ ){
    C( 1, 3 ) += A( 1, p ) * B( p, 3 );
  }

  /* Third row */
  //  AddDot( k, &A( 2, 0 ), lda, &B( 0, 0 ), &C( 2, 0 ) );
  for ( p=0; p<k; p++ ){
    C( 2, 0 ) += A( 2, p ) * B( p, 0 );
  }
  //  AddDot( k, &A( 2, 0 ), lda, &B( 0, 1 ), &C( 2, 1 ) );
  for ( p=0; p<k; p++ ){
    C( 2, 1 ) += A( 2, p ) * B( p, 1 );
  }
  //  AddDot( k, &A( 2, 0 ), lda, &B( 0, 2 ), &C( 2, 2 ) );
  for ( p=0; p<k; p++ ){
    C( 2, 2 ) += A( 2, p ) * B( p, 2 );
  }
```

```
  //  AddDot( k, &A( 2, 0 ), lda, &B( 0, 3 ), &C( 2, 3 ) );
  for ( p=0; p<k; p++ ){
    C( 2, 3 ) += A( 2, p ) * B( p, 3 );
  }

  /* Fourth row */
  //  AddDot( k, &A( 3, 0 ), lda, &B( 0, 0 ), &C( 3, 0 ) );
  for ( p=0; p<k; p++ ){
    C( 3, 0 ) += A( 3, p ) * B( p, 0 );
  }
  //  AddDot( k, &A( 3, 0 ), lda, &B( 0, 1 ), &C( 3, 1 ) );
  for ( p=0; p<k; p++ ){
    C( 3, 1 ) += A( 3, p ) * B( p, 1 );
  }
  //  AddDot( k, &A( 3, 0 ), lda, &B( 0, 2 ), &C( 3, 2 ) );
  for ( p=0; p<k; p++ ){
    C( 3, 2 ) += A( 3, p ) * B( p, 2 );
  }
  //  AddDot( k, &A( 3, 0 ), lda, &B( 0, 3 ), &C( 3, 3 ) );
  for ( p=0; p<k; p++ ){
    C( 3, 3 ) += A( 3, p ) * B( p, 3 );
  }
}
```



No performance benefit yet

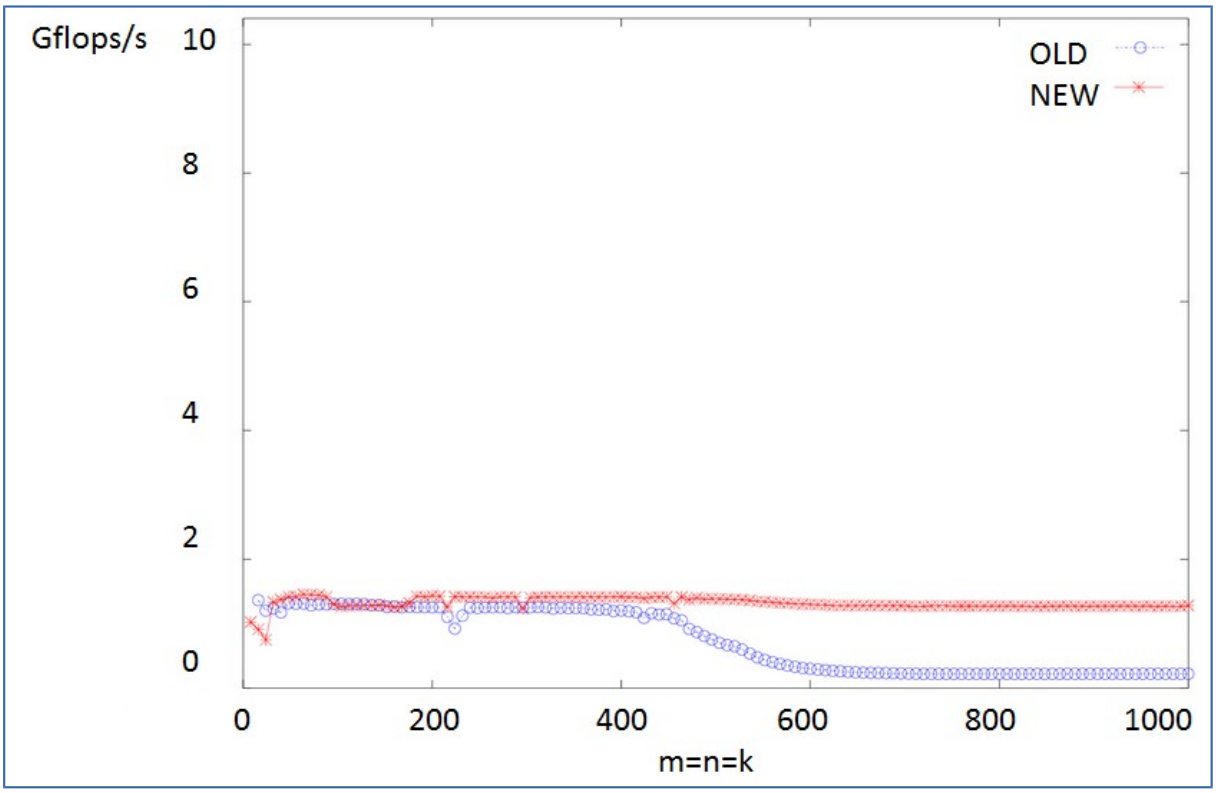
```
void AddDot4x4( int k, double *a, int lda,
                       double *b, int ldb,
                       double *c, int ldc )
{
  int p;

  /* First row */
  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 0 ), &C( 0, 0 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 0 ) += A( 0, p ) * B( p, 0 );
  }
  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 1 ), &C( 0, 1 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 1 ) += A( 0, p ) * B( p, 1 );
  }
  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 2 ), &C( 0, 2 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 2 ) += A( 0, p ) * B( p, 2 );
  }
  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 3 ), &C( 0, 3 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 3 ) += A( 0, p ) * B( p, 3 );
  }

  /* Second row */
  //  AddDot( k, &A( 1, 0 ), lda, &B( 0, 0 ), &C( 1, 0 ) );
  for ( p=0; p<k; p++ ){
    C( 1, 0 ) += A( 1, p ) * B( p, 0 );
  }
  //  AddDot( k, &A( 1, 0 ), lda, &B( 0, 1 ), &C( 1, 1 ) );
  for ( p=0; p<k; p++ ){
    C( 1, 1 ) += A( 1, p ) * B( p, 1 );
  }
  //  AddDot( k, &A( 1, 0 ), lda, &B( 0, 2 ), &C( 1, 2 ) );
  for ( p=0; p<k; p++ ){
    C( 1, 2 ) += A( 1, p ) * B( p, 2 );
  }
  //  AddDot( k, &A( 1, 0 ), lda, &B( 0, 3 ), &C( 1, 3 ) );
  for ( p=0; p<k; p++ ){
    C( 1, 3 ) += A( 1, p ) * B( p, 3 );
  }

  /* Third row */
  //  AddDot( k, &A( 2, 0 ), lda, &B( 0, 0 ), &C( 2, 0 ) );
  for ( p=0; p<k; p++ ){
    C( 2, 0 ) += A( 2, p ) * B( p, 0 );
  }
  //  AddDot( k, &A( 2, 0 ), lda, &B( 0, 1 ), &C( 2, 1 ) );
  for ( p=0; p<k; p++ ){
    C( 2, 1 ) += A( 2, p ) * B( p, 1 );
```

```
  }
  //  AddDot( k, &A( 2, 0 ), lda, &B( 0, 2 ), &C( 2, 2 ) );
  for ( p=0; p<k; p++ ){
    C( 2, 2 ) += A( 2, p ) * B( p, 2 );
  }
  //  AddDot( k, &A( 2, 0 ), lda, &B( 0, 3 ), &C( 2, 3 ) );
  for ( p=0; p<k; p++ ){
    C( 2, 3 ) += A( 2, p ) * B( p, 3 );
  }

  /* Fourth row */
  //  AddDot( k, &A( 3, 0 ), lda, &B( 0, 0 ), &C( 3, 0 ) );
  for ( p=0; p<k; p++ ){
    C( 3, 0 ) += A( 3, p ) * B( p, 0 );
  }
  //  AddDot( k, &A( 3, 0 ), lda, &B( 0, 1 ), &C( 3, 1 ) );
  for ( p=0; p<k; p++ ){
    C( 3, 1 ) += A( 3, p ) * B( p, 1 );
  }
  //  AddDot( k, &A( 3, 0 ), lda, &B( 0, 2 ), &C( 3, 2 ) );
  for ( p=0; p<k; p++ ){
    C( 3, 2 ) += A( 3, p ) * B( p, 2 );
  }
  //  AddDot( k, &A( 3, 0 ), lda, &B( 0, 3 ), &C( 3, 3 ) );
  for ( p=0; p<k; p++ ){
    C( 3, 3 ) += A( 3, p ) * B( p, 3 );
  }
}
```

We merge all the loops (now we have only one loop)

```
  for ( p=0; p<k; p++ ){
    /* First row */
    C( 0, 0 ) += A( 0, p ) * B( p, 0 );
    C( 0, 1 ) += A( 0, p ) * B( p, 1 );
    C( 0, 2 ) += A( 0, p ) * B( p, 2 );
    C( 0, 3 ) += A( 0, p ) * B( p, 3 );

    /* Second row */
    C( 1, 0 ) += A( 1, p ) * B( p, 0 );
    C( 1, 1 ) += A( 1, p ) * B( p, 1 );
    C( 1, 2 ) += A( 1, p ) * B( p, 2 );
    C( 1, 3 ) += A( 1, p ) * B( p, 3 );

    /* Third row */
    C( 2, 0 ) += A( 2, p ) * B( p, 0 );
    C( 2, 1 ) += A( 2, p ) * B( p, 1 );
    C( 2, 2 ) += A( 2, p ) * B( p, 2 );
```

```
    C( 2, 3 ) += A( 2, p ) * B( p, 3 );

    /* Fourth row */
    C( 3, 0 ) += A( 3, p ) * B( p, 0 );
    C( 3, 1 ) += A( 3, p ) * B( p, 1 );
    C( 3, 2 ) += A( 3, p ) * B( p, 2 );
    C( 3, 3 ) += A( 3, p ) * B( p, 3 );
  }
}
```

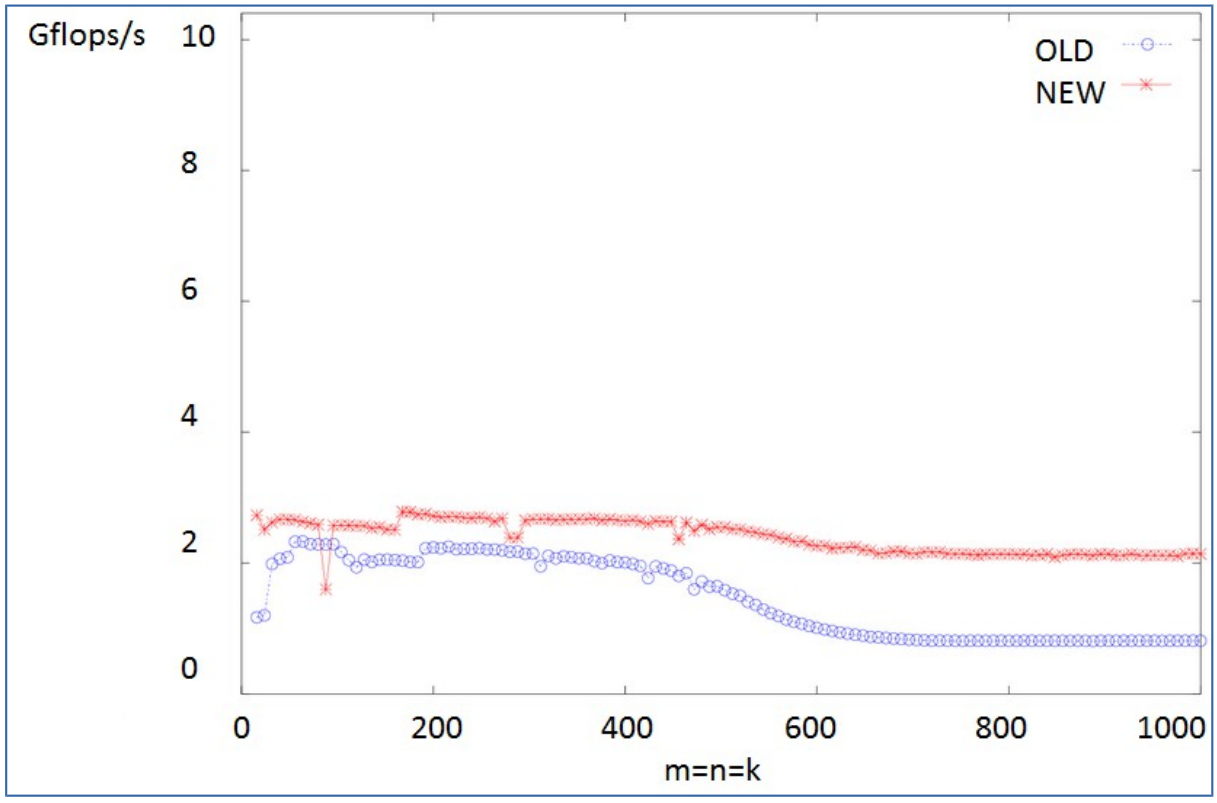


Next step - we accumulate in registers

```
void AddDot4x4( int k, double *a, int lda,
                       double *b, int ldb,
                       double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A

     In this version, we accumulate in registers and put A( 0,
p ) in a register */
```

```
  int p;
    /* hold contributions to
       C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 )
       C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 )
       C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 )
       C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 )   */
  register double
       c_00_reg,   c_01_reg,   c_02_reg,   c_03_reg,
       c_10_reg,   c_11_reg,   c_12_reg,   c_13_reg,
       c_20_reg,   c_21_reg,   c_22_reg,   c_23_reg,
       c_30_reg,   c_31_reg,   c_32_reg,   c_33_reg;

    /* hold
       A( 0, p )
       A( 1, p )
       A( 2, p )
       A( 3, p ) */
  register double
       a_0p_reg,
       a_1p_reg,
       a_2p_reg,
       a_3p_reg;

  c_00_reg = 0.0;   c_01_reg = 0.0;
       c_02_reg = 0.0;   c_03_reg = 0.0;
  c_10_reg = 0.0;   c_11_reg = 0.0;
       c_12_reg = 0.0;   c_13_reg = 0.0;
  c_20_reg = 0.0;   c_21_reg = 0.0;
       c_22_reg = 0.0;   c_23_reg = 0.0;
  c_30_reg = 0.0;   c_31_reg = 0.0;
       c_32_reg = 0.0;   c_33_reg = 0.0;

  for ( p=0; p<k; p++ ){
    a_0p_reg = A( 0, p );
    a_1p_reg = A( 1, p );
    a_2p_reg = A( 2, p );
    a_3p_reg = A( 3, p );

    /* First row */
    c_00_reg += a_0p_reg * B( p, 0 );
    c_01_reg += a_0p_reg * B( p, 1 );
    c_02_reg += a_0p_reg * B( p, 2 );
    c_03_reg += a_0p_reg * B( p, 3 );

    /* Second row */
    c_10_reg += a_1p_reg * B( p, 0 );
    c_11_reg += a_1p_reg * B( p, 1 );
    c_12_reg += a_1p_reg * B( p, 2 );
    c_13_reg += a_1p_reg * B( p, 3 );
```

```
    /* Third row */
    c_20_reg += a_2p_reg * B( p, 0 );
    c_21_reg += a_2p_reg * B( p, 1 );
    c_22_reg += a_2p_reg * B( p, 2 );
    c_23_reg += a_2p_reg * B( p, 3 );

    /* Four row */
    c_30_reg += a_3p_reg * B( p, 0 );
    c_31_reg += a_3p_reg * B( p, 1 );
    c_32_reg += a_3p_reg * B( p, 2 );
    c_33_reg += a_3p_reg * B( p, 3 );
  }

  C( 0, 0 ) += c_00_reg;   C( 0, 1 ) += c_01_reg;
       C( 0, 2 ) += c_02_reg;   C( 0, 3 ) += c_03_reg;
  C( 1, 0 ) += c_10_reg;   C( 1, 1 ) += c_11_reg;
       C( 1, 2 ) += c_12_reg;   C( 1, 3 ) += c_13_reg;
  C( 2, 0 ) += c_20_reg;   C( 2, 1 ) += c_21_reg;
       C( 2, 2 ) += c_22_reg;   C( 2, 3 ) += c_23_reg;
  C( 3, 0 ) += c_30_reg;   C( 3, 1 ) += c_31_reg;
       C( 3, 2 ) += c_32_reg;   C( 3, 3 ) += c_33_reg;
}
```



The performance benefits from the fact that we use (regular) registers for the 4x4 block of C and the elements of the current 4x1 column of A. Notice that we are using more regular registers than there exist, so it is anyone's guess what the compiler does with that.

Use pointers for accessing B

```
void AddDot4x4( int k, double *a, int lda,  double *b, int ldb,
double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A
     In this version, we use pointer to track
     where in four columns of B we are */

  int p;
  register double
    /* hold contributions to
       C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 )
       C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 )
       C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 )
       C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 )   */
       c_00_reg,   c_01_reg,   c_02_reg,   c_03_reg,
       c_10_reg,   c_11_reg,   c_12_reg,   c_13_reg,
       c_20_reg,   c_21_reg,   c_22_reg,   c_23_reg,
       c_30_reg,   c_31_reg,   c_32_reg,   c_33_reg,
    /* hold
       A( 0, p )
       A( 1, p )
       A( 2, p )
       A( 3, p ) */
       a_0p_reg,
       a_1p_reg,
       a_2p_reg,
       a_3p_reg;
  double
    /* Point to the current elements in the four columns of B
*/
    *b_p0_pntr, *b_p1_pntr, *b_p2_pntr, *b_p3_pntr;

  b_p0_pntr = &B( 0, 0 );
  b_p1_pntr = &B( 0, 1 );
  b_p2_pntr = &B( 0, 2 );
  b_p3_pntr = &B( 0, 3 );

  c_00_reg = 0.0;   c_01_reg = 0.0;
      c_02_reg = 0.0;   c_03_reg = 0.0;
  c_10_reg = 0.0;   c_11_reg = 0.0;
      c_12_reg = 0.0;   c_13_reg = 0.0;
  c_20_reg = 0.0;   c_21_reg = 0.0;
      c_22_reg = 0.0;   c_23_reg = 0.0;
  c_30_reg = 0.0;   c_31_reg = 0.0;
      c_32_reg = 0.0;   c_33_reg = 0.0;
```

```
  for ( p=0; p<k; p++ ){
    a_0p_reg = A( 0, p );
    a_1p_reg = A( 1, p );
    a_2p_reg = A( 2, p );
    a_3p_reg = A( 3, p );

    /* First row */
    c_00_reg += a_0p_reg * *b_p0_pntr;
    c_01_reg += a_0p_reg * *b_p1_pntr;
    c_02_reg += a_0p_reg * *b_p2_pntr;
    c_03_reg += a_0p_reg * *b_p3_pntr;

    /* Second row */
    c_10_reg += a_1p_reg * *b_p0_pntr;
    c_11_reg += a_1p_reg * *b_p1_pntr;
    c_12_reg += a_1p_reg * *b_p2_pntr;
    c_13_reg += a_1p_reg * *b_p3_pntr;

    /* Third row */
    c_20_reg += a_2p_reg * *b_p0_pntr;
    c_21_reg += a_2p_reg * *b_p1_pntr;
    c_22_reg += a_2p_reg * *b_p2_pntr;
    c_23_reg += a_2p_reg * *b_p3_pntr;

    /* Four row */
    c_30_reg += a_3p_reg * *b_p0_pntr++;
    c_31_reg += a_3p_reg * *b_p1_pntr++;
    c_32_reg += a_3p_reg * *b_p2_pntr++;
    c_33_reg += a_3p_reg * *b_p3_pntr++;
  }

  C( 0, 0 ) += c_00_reg;   C( 0, 1 ) += c_01_reg;
      C( 0, 2 ) += c_02_reg;   C( 0, 3 ) += c_03_reg;
  C( 1, 0 ) += c_10_reg;   C( 1, 1 ) += c_11_reg;
      C( 1, 2 ) += c_12_reg;   C( 1, 3 ) += c_13_reg;
  C( 2, 0 ) += c_20_reg;   C( 2, 1 ) += c_21_reg;
      C( 2, 2 ) += c_22_reg;   C( 2, 3 ) += c_23_reg;
  C( 3, 0 ) += c_30_reg;   C( 3, 1 ) += c_31_reg;
      C( 3, 2 ) += c_32_reg;   C( 3, 3 ) += c_33_reg;
}
```