Week B

Ladies and Gentlemen, Start Your Engines!

B.1 Opener

B.1.1 Launch ☛ to edX

Optimizing a code is like tinkering on a Formula One race car. Addicting!

B.1.2 Outline Week 1 ☛ to edX

B.1.3 What you will learn ☛ to edX

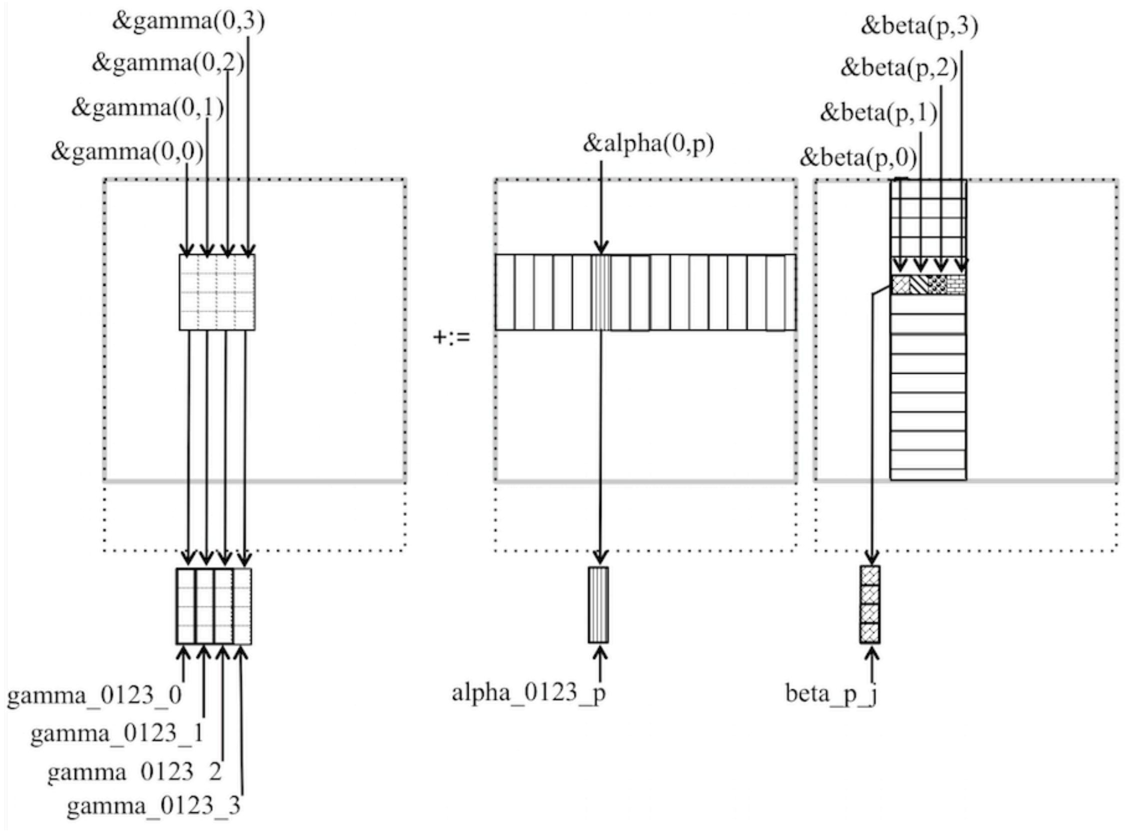


Figure B.1: Illustration of how register variables are used to implement the update of a 4×4 submatrix of C. (Repeat of Figure **??**.)

B.2 A Microkernel

B.2.1 Utilizing Registers

In Section **??**, we introduced the fact that modern architectures have vector registers and then showed how to (somewhat) optimize the update of a 4×4 submatrix of C. The picture that captured this is given again, in Figure B.1, with the loop for the routine previously given in Figure A.2 now in Figure B.2.

Some observations:

- Four vector registers are used for the submatrix of C. Once loaded, those values remain in the registers for the duration of the computation, until they are finally written out once they have been completely updated. It is important to keep values of C in registers that long, because values of C are read as well as written, unlike values from A and B, which are read but not written.

- The block is 4×4. In general, we can view the block as being $m_r \times n_r$, where in this case $m_r = n_r = 4$. If we assume we will use r_c registers for elements of the submatrix of C, what

```
for ( int p=0; p<k; p++ ){
  /* load alpha( 0:3, p ) 8?
  __m256d alpha_0123_p = _mm256_loadu_pd( &alpha( 0,p ) );

  /* load beta( p, 0 ) */
  __m256d beta_p_j = _mm256_broadcast_sd( &beta( p, 0) );
  /* update gamma( 0:3, 0 ) */
  gamma_0123_0 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_0 );

  /* load beta( p, 1 ) */
  __m256d beta_p_j = _mm256_broadcast_sd( &beta( p, 1) );
  /* update gamma( 0:3, 1 ) */
  gamma_0123_1 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_1 );

  /* load beta( p, 2 ) */
  __m256d beta_p_j = _mm256_broadcast_sd( &beta( p, 2) );
  /* update gamma( 0:3, 2 ) */
  gamma_0123_2 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_2 );

   /* load beta( p, 3 ) */
  __m256d beta_p_j = _mm256_broadcast_sd( &beta( p, 3) );
  /* update gamma( 0:3, 2 ) */
  gamma_0123_3 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_3 );
}
```

github/LAFF–On–PfHPC/Assignments/WeekA/C/GemmIntrinsicsKernel_m4xn4Part.c

Figure B.2: Loop for computing $C := AB + C$ where C is 4×4 (See also Figure A.2.)

should m_r and n_r be, given the constraint that $m_r \times n_r = r_c$?

- Given that some registers need to be used for elements of A and B, how many registers should be used for elements of each of the matrices?

In this section, we hope to address some of these.

B.2.2 It's hip to be square

One of the fundamental principles behind high performance is that when data is moved from one layer of memory to another, one needs to amortize the cost of that data movement over as many computations as possible. In our discussion so far, we have only two layers of memory: (vector) registers and main memory. Later we will generalize our observations to many such layers (namely, multiple cache memories).

Let us focus on the current approach:

- Four vector registers are loaded with a 4×4 submatrix of C. In our picture, we call these registers `gamma_0123_0` through `gamma_0123_3`. The cost of loading and unloading this data is negligible if the k dimension of the problem is large.

- One vector register is loaded with the current column of A with four elements. In our picture, we call this register `alpha_0123_p`. The cost of this load is amortized over 32 floating point operations: a multiply and an add with each of the sixteen elements of the submatrix of C.

- A vector register is loaded with and elements of the current row of B. While in our previous explanation each element is duplicated in a distinct vector register, in fact they could all use the same register since the new value of B is not loaded until after the previous one has been used and is no longer needed. It is not quite this simple, but let's move on with our explanation under this assumption for now. We can call the this register `beta_p_j` The load of the element is amortized over eight floating point operations.

If we generalize this, under the assumption that a vector register can hold 4 floats and m_r is a multiple of 4, we can work with a $m_r \times n_r$ submatrix of C, using $r_C = (m_r/4) \times n_r$ vector registers, and requiring $m_r/4$ vector registers for the elements of A and another $m_r/4$ vector registers for the duplicated element of B. Ignoring the cost of loading the submatrix of C, this would then require

$$m_r + n_r$$

floating point number loads from slow memory, of m_r elements of A and n_r elements of B, which are amortized over $2 \times m_r \times n_r$ double precision floating point operations (flops), for each iteration indexed by p.

What we want is to amortize the cost of the loads over as many computations as possible. It can be shown that if $r_c = m_r \times n_r$ is kept constant, then the ratio $2m_r \times n_r/(m_r + n_r)$ (the ratio of computation to communication between fast memory and slow memory) is maximized when $m_r \approx n_r$. In other words, when the submatrix of C is roughly square. Here we say "roughly" because there are other factors to be taken into account, such as the vector register length (which at least one of m_r and n_r must be a multiple of) and differences in the cost of various vector operations.

Homework B.2.2.1 Assume that x and y are both positive and that $x \times y = C$, where C is a constant. Find the choices for x and y that maximize $(x \times y)/(x + y)$.

☛ SEE ANSWER

☛ DO EXERCISE ON edX

B.2.3 Using the available vector registers

In our discussion before, we are only using four vector registers for matrix C and one vector registers for each of A and B, for a total of six vector registers. The architecture we are targeting so for, an Intel Haswell processor, has sixteen vector registers (per core). The larger m_r and n_r, the better the ratio $m_r \times n_r/(m_r + n_r)$ if $m_r \approx n_r$. The constraint is that $m_r \times n_r \leq 16 - (r_A + r_B)$ where r_A and r_B are the number of registers used for elements of r_A and r_B, respectively. This leads us to ask how to use all sixteen vector registers effectively.

While there is an analytical (approximation to) an answer to this question, the details of this go beyond this course. Let's discuss some possibilities, including an admittedly somewhat naive analysis.

- $m_R \times n_R = 4 \times 4$. We have analyzed this. It uses six of the sixteen registers. For each k, the ratio of computation to communication is at best $4 \times 4/(4+4) = 2$.

- $m_R \times n_R = 8 \times 4$. This use eight registers (half of those available) for C, at least two registers for A, and at least one register for B. Now the ratio becomes $8 \times 4/(8+4) = 32/12$.

- $m_R \times n_R = 4 \times 8$. This also use eight registers (half of those available) for C, at least one register for A, and at least one register for B. The ratio becomes $8 \times 4/(8+4) = 32/12 \approx 2.7$.

- $m_R \times n_R = 8 \times 6$. This uses twelve registers (three quarters of those available) for C, at least two registers for A, and at least one register for B. The ratio becomes $8 \times 6/(8+6) = 48/14 \approx 3.4$.

- $m_R \times n_R = 6 \times 8$. This uses twelve registers (three quarters of those available) for C. The problem is that the natural extension of the scheme we used for a 4×4 kernel would be to use one and a half vector registers for the elements of A. There are alternatives to this, but let's discuss those later.

- $m_R \times n_R = 4 \times 12$.

- $m_R \times n_R = 12 \times 4$.

- $m_R \times n_R = 8 \times 7$. One could, conceivably, use a kernel with an odd number of columns in the submatrix of C. Let's discuss that later as well.

- $m_R \times n_R = 8 \times 8$. This would use all registers for C, leaving no registers for A or B. That just doesn't work.

The bottom line: There are many choices and one might want to explore some subset of these.

Homework B.2.3.1 In Figures B.3-B.4, we give a framework for implementing a $m_r \times n_r$ kernel, instantiated for the case where $m_r = n_r = 4$. It can be found in

```
WeekC/C/GemmJI_Kernel_MRxNR.c.
```

executed with by typing

make test_GemmJI_Kernel_MRxNR

in that directory. The resulting performance can be viewed with Live Script `WeekC/C/ShowPerformance.mlx`. Modify it to implement your choice of kernel (meaning you can pick m_r and n_r.

☛ SEE ANSWER

☛ DO EXERCISE ON edX

```
#define alpha( i,j ) A[ (j)*ldA + (i) ]   // map alpha( i,j ) to array A
#define beta( i,j )  B[ (j)*ldB + (i) ]   // map beta( i,j ) to array B
#define gamma( i,j ) C[ (j)*ldC + (i) ]   // map gamma( i,j ) to array C
#define MR 4
#define NR 4

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

void GemmJI_Kernel_MRxNR( int m, int n, int k, double *A, int ldA,
        double *B, int ldB, double *C, int ldC )
{
  for ( int j=0; j<n; j+=NR ) /* n is assumed to be a multiple of NR */
    for ( int i=0; i<m; i+=MR ) /* m is assumed to be a multiple of MR */
      GemmIntrinsicsKernel_MRxNR
        ( k, &alpha( i,0 ), ldA, &beta( 0,j ), ldB, &gamma( i,j ), ldC );
}
```

github/LAFF–On–PfHPC/Assignments/WeekB/C/GemmJI_Kernel_MRxNR.c

Figure B.3: General framework for calling a kernel, instantiated for the case where $m_r = n_r = 4$. (Continued in Figure B.4.)

B.2.4 Overlapping communication with computation

In the last subsection, we discussed amortizing the cost of loading elements of A and B as if the loading of such elements cannot overlap with computation. In face, in a typical core, one can compute while data (for future computation) is being fetched from (some layer of) memory. What this means is that one wants to pick m_r and n_r to maximize the amount of computation performed per element of A and/or B so as to best overlap the cost of bringing in the next such data. What this, in turn, means is that one needs more than the registers that are used to store the elements of A and B. One also needs registers into which to *prefetch* future such elements, and one needs to expose such registers in the code. Fortunately, much of this is automatically done by a decent compiler.

B.2.5 Can this work?

Notice that we have analyzed that we must hide the loading of elements of A and B with, at best 3-4 floating point operations per such element. The "latency" from main memory (the time it takes for a double precision number to be fetched from main memory) is more around the time it takes to perform 100 floating point operations. Fortunately, modern architectures are equipped with multiple layers of cache memories that have a lower latency.

In Figure B.5 (top-left), we report the performance attained by our implementation for Homework B.2.3.1, with $m_r = n_r = 4$. What we notice is that when the matrices are small enough to roughly fit in the L2 cache, performance is somewhat respectable, relative to a high-performance reference implementation. The performance drops when they fit in the (slower) L3 cache, and performance drops again once the matrices are large enough that they don't fit in the L3 cache.

```
#include<immintrin.h>

void GemmIntrinsicsKernel_MRxNR( int k, double *A, int ldA,
         double *B, int ldB, double *C, int ldC )
{
  __m256d gamma_0123_0 = _mm256_loadu_pd( &gamma( 0,0 ) );
  __m256d gamma_0123_1 = _mm256_loadu_pd( &gamma( 0,1 ) );
  __m256d gamma_0123_2 = _mm256_loadu_pd( &gamma( 0,2 ) );
  __m256d gamma_0123_3 = _mm256_loadu_pd( &gamma( 0,3 ) );

  __m256d beta_p_j;

  for ( int p=0; p<k; p++ ){
    /* load alpha( 0:3, p ) */
    __m256d alpha_0123_p = _mm256_loadu_pd( &alpha( 0,p ) );
    /* load beta( p, 0 ); update gamma( 0:3, 0 ) */
    beta_p_j = _mm256_broadcast_sd( &beta( p,0) );
    gamma_0123_0 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_0 );
    /* load beta( p, 1 ); update gamma( 0:3, 1 ) */
    beta_p_j = _mm256_broadcast_sd( &beta( p,1) );
    gamma_0123_1 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_1 );
    /* load beta( p, 2 ); update gamma( 0:3, 2 ) */
    beta_p_j = _mm256_broadcast_sd( &beta( p,2) );
    gamma_0123_2 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_2 );
    /* load beta( p, 3 ); update gamma( 0:3, 3 ) */
    beta_p_j = _mm256_broadcast_sd( &beta( p,3) );
    gamma_0123_3 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_3 );
  }

  /* Store the updated results */
  _mm256_storeu_pd( &gamma(0,0), gamma_0123_0 );
  _mm256_storeu_pd( &gamma(0,1), gamma_0123_1 );
  _mm256_storeu_pd( &gamma(0,2), gamma_0123_2 );
  _mm256_storeu_pd( &gamma(0,3), gamma_0123_3 );
}
```

github/LAFF–On–PfHPC/Assignments/WeekB/C/GemmJI_Kernel_MRxNR.c

Figure B.4: (Continued) General framework for calling a kernel, instantiated for the case where $m_r = n_r = 4$.

B.3 Around the Microkernel

B.3.1 The memory hierarchy

The inconvenient truth is that floating point computations can be performed very fast while bringing data in from main memory is relatively slow. How slow? On a typical architecture it takes two orders of magnitude more time to bring a floating point number in from main memory than it takes

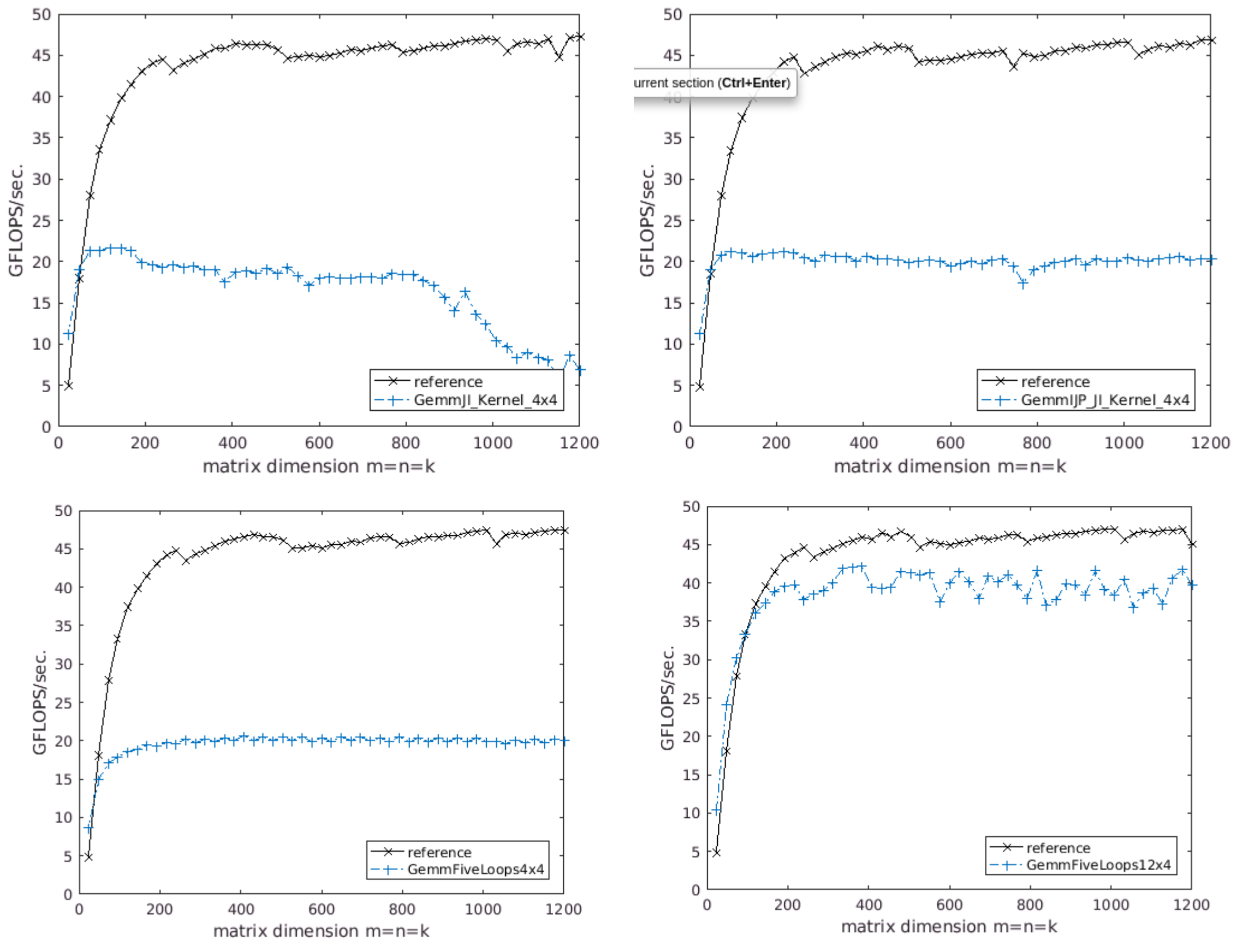


Figure B.5: Top-left: Our performance for the exercise in Homework B.2.3.1, where we created a double loop around the computation of 4×4 submatrices of C. Top-right: Our performance for the exercise in Homework B.2.3.1, where we partitioned the matrices into (roughly) square blocks, for each of which we then called the routine used in the top-left graph. Bottom-left: An implementation of the algorithm in Figure B.10, utilizing the micro-kernel (coded with vector intrinsic function) for $m_r \times n_r = 4 \times 4$. Bottom-right: Same routine as for the bottom-left graph, but with a $m_r \times n_r = 12 \times 4$ micro-kernel.

to compute with it.

The reason why main memory is slow is relatively simple: there is not enough room on a chip for the large memories that we are accustomed to, and hence they are off chip. The mere distance creates a latency for retrieving the data. This could then be offset by retrieving a lot of data simultaneously. Unfortunately there are inherent bandwidth limitations: there are only so many pins that can connect the central processing unit with main memory.

To overcome this limitation, a modern processor has a hierarchy of memories. We have already encountered the two extremes: registers and main memory. In between, there are smaller but faster *cache memories*. These cache memories are on-chip and hence do not carry the same latency

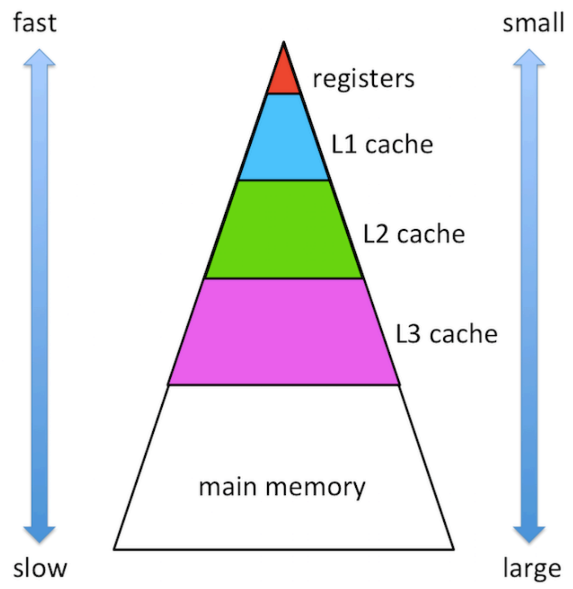


Figure B.6: Illustration of the memory hierarchy.

as does main memory and also can achieve greater bandwidth. The hierarchical nature of these memories is often depicted as a pyramid as illustrated in Figure B.6. To put things in perspective: We have discussed that the Haswell processor has sixteen vector processors that can store 64 double precision floating point numbers (floats). Close to the CPU it has a 32Kbytes level-1 cache (L1 cache) that can thus store 4K floats. Somewhat further it has a 256Kbytes level-2 cache (L2 cache). Further away yet, but still on chip, it has a 8Mbytes level-3 cache (L3 cache) that can hold ?? floats.

In the below discussion, we will pretend that one can place data in a specific cache and keep it there for the duration of computations. In fact, caches retain data using some replacement policy that evicts data that has not been recently used. By carefully ordering computations, we can encourage data to remain in cache, which is what happens in practice.

B.3.2 Maintaining performance beyond the L2 cache

Let us, for the sake of argument, start with the following assumptions:

- The architecture has one level of cache.

- To attain high performance, the kernel we wrote must use data that is stored in that cache. In other words, the $m_r \times n_r$ submatrix of C, the $m_r \times k_c$ submatrix of A, and the $k_c \times n_r$ submatrix of B must all reside in that cache. Here parameter k_c will become clear as our explanation proceeds.

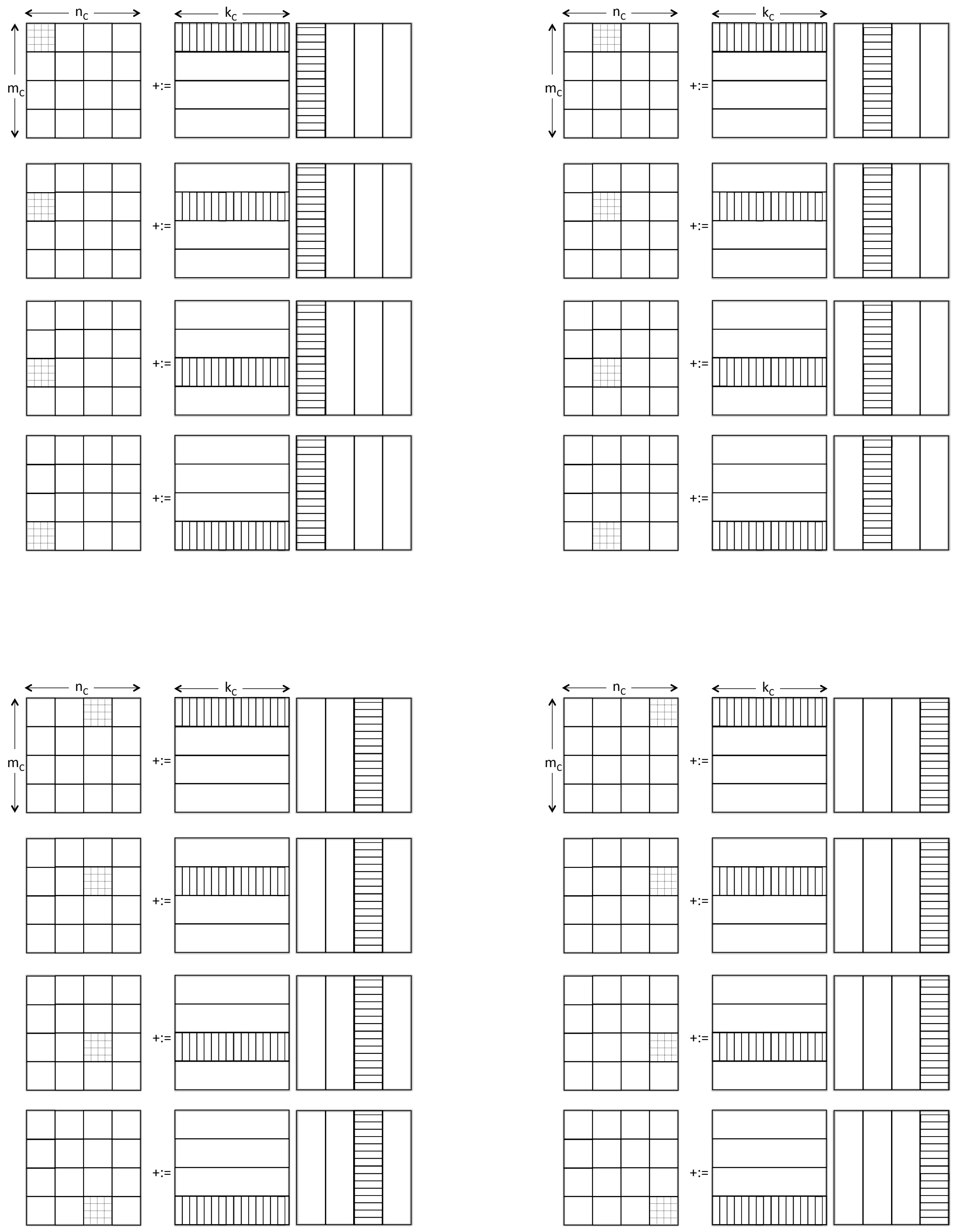


Figure B.7: Illustration of computation in the cache with one block from each of A, B, and C.

A naive approach partitions C, A, and B into (roughly) square blocks:

$$C = \left(\begin{array}{c|c|c|c} C_{0,0} & C_{0,1} & \cdots & C_{0,N-1} \\ \hline C_{1,0} & C_{1,1} & \cdots & C_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline C_{M-1,0} & C_{M,1} & \cdots & C_{M-1,N-1} \end{array} \right), A = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,K-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,K-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{M-1,0} & A_{M,1} & \cdots & A_{M-1,K-1} \end{array} \right),$$

and

$$B = \left(\begin{array}{c|c|c|c} B_{0,0} & B_{0,1} & \cdots & B_{0,N-1} \\ \hline B_{1,0} & B_{1,1} & \cdots & B_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline B_{K-1,0} & B_{K,1} & \cdots & B_{K-1,N-1} \end{array} \right),$$

where

- $C_{i,j}$ is $m_c \times n_c$,
- $A_{i,p}$ is $m_c \times k_c$, and
- $B_{p,j}$ is $k_c \times n_c$.

Then

$$C_{i,j} := \sum_{p=0}^{K-1} A_{i,p} B_{p,j} + C_{i,j},$$

which can be written as the triple-nested loop

```
**for** $I := 0, \ldots, M-1$
  **for** $J := 0, \ldots, N-1$
      **for** $P := 0, \ldots, K-1$
          $C_{i,j} := A_{i,p} B_{p,j} + B_{i,j}$
      **end**
    **end**
**end**
```

(or any of the other loop orderings.)

If we choose m_c, n_c, and k_c such that $C_{i,j}$, $A_{i,p}$, and $B_{p,j}$ all fit in the cache, then we meet our conditions. We can then compute $C_{i,j} := A_{i,p} B_{p,j} + C_{i,j}$ by bringing these blocks into cache and computing with them before writing out the result, as before.

```c
#define alpha( i,j ) A[ (j)*ldA + (i) ]   // map alpha( i,j ) to array A
#define beta( i,j )  B[ (j)*ldB + (i) ]   // map beta( i,j ) to array B
#define gamma( i,j ) C[ (j)*ldC + (i) ]   // map gamma( i,j ) to array C

#define min( x, y ) ( (x) < (y) ? x : y )

#define MR 4
#define NR 4

#define MC 128
#define NC 128
#define KC 128

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

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

void GemmIJP_JI_Kernel_MRxNR( int m, int n, int k, double *A, int ldA,
        double *B, int ldB, double *C, int ldC )
{
  int ib, jb, pb;

  for ( int i=0; i<m; i+=MC ) {
    ib = min( MC, m-i );        /* Last block may not be a full block */
    for ( int j=0; j<n; j+=NC ) {
      jb = min( NC, n-j );        /* Last block may not be a full block */
      for ( int p=0; p<k; p+=KC ) {
        pb = min( KC, k-p );        /* Last block may not be a full block */

        GemmJI_Kernel_MRxNR( ib, jb, pb,
          &alpha( i,p ), ldA, &beta( p,j ), ldB, &gamma( i,j ), ldC );
      }
    }
  }
}
```

github/LAFF–On–PfHPC/Assignments/WeekB/C/GemmIJP_JI_Kernel_MRxNR.c

Figure B.8: Triple loop around $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$ as implemented by `GemmIJP_JI_Kernel_MRxNR`.

Homework B.3.2.1 In Figure B.8, we give an IJP loop ordering around the computation of $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$, which itself is implemented by `GemmJI_Kernel_MRxNR`. It can be found in

`WeekC/C/GemmIJP_JI_Kernel_MRxNR.c.`

executed by typing

```
make test_GemmIJP_JI_Kernel_MRxNR
```

in that directory. The resulting performance can be viewed with Live Script `WeekC/C/ShowPerformance.mlx`. Modify it to implement your choice of kernel (meaning you can pick m_r and n_r. Next, modify it to use the kernel you wrote for Homework B.2.3.1.

☛ SEE ANSWER

☛ DO EXERCISE ON edX

The performance attained by our implementation is shown in Figure B.5 (top-right).

We illustrate the computation with one set of three submatrices (one per matrix A, B, and C) in Figure B.7 for the case where $m_r = n_r = 4$ and $m_c = n_c = k_c = 4m_r$. What we notice is that the $m_r \times n_r$ submatrices of $C_{i,j}$ are not reused once updated. This means that at any given time only one such matrix needs to be in the cache memory. Similarly, a panel of $B_{p,j}$ is not reused and hence only one such panel needs to be in cache. It is submatrix $A_{i,p}$ that must remain in cache during the entire computation. By not keeping the entire submatrices $C_{i,j}$ and $B_{p,j}$ in the cache memory, the size of the submatrix $A_{i,p}$ can be increased, which can be expected to improve performance.

To summarize, our insights suggest

1. Bring an $m_c \times k_c$ submatrix of A into the cache memory, at a cost of $m_c \times k_c$ memops. This cost is amortized over $2m_cn_ck_c$ flops for a ratio of $2m_cn_ck_c/(m_ck_c) = 2n_c$. The larger n_c, the better.

2. The cost of reading an $k_c \times n_r$ submatrix of B is amortized over $2m_cn_rk_c$ flops, for a ratio of $2m_cn_rk_c/(2m_cn_r) = m_c$. Obviously, the greater m_c, the better.

3. The cost of reading and writing an $m_r \times n_r$ submatrix of C is now amortized over $2m_rn_rk_c$ flops, for a ratio of $2m_rn_rk_c/(2m_rn_r) = k_c$. Obviously, the greater k_c, the better.

Items 2 and 3 suggest that $m_c \times k_c$ submatrix $A_{i,p}$ be roughly square.

If we revisit the performance data plotted in Figure A.6.3.1, we point out that performance starts to first drop when $m = n = k = 128$. Storing *one* matrix of size 128×128 requires 128Kbytes, which is exactly half the size of the L2 cache for a Haswell core. Our discussion about how only one matrix block needs to be stored in "the cache" explains why this was observed (given that a naive analysis would have suggested that three matrices need to remain in cache).

In our future discussions, we will use the following terminology:

- A $m_r \times n_r$ submatrix of C that is begin updated we will call a *micro-tile*.
- The $m_r \times k_C$ submatrix of A and $k_C \times n_r$ submatrix of B we will call *micro-panels*.
- The routine that updates a micro-tile by multiplying two micro-panels we will call the micro-kernel.

B.3.3 Blocking for multiple caches

We now describe one way for blocking for multiple levels of cache. While some details still remain, this brings us very close to how matrix-matrix multiplication is implemented in practice in libraries that are widely used.

Each nested box represents a single loop that partitions one of the three dimensions (m, n, or k). The submatrices $C_{i,j}$, $A_{i,p}$, and $B_{p,j}$ are those already discussed in Section B.3.2.

- The roughly square matrix $A_{i,p}$ is "placed" in the L2 cache. So, if data from $A_{i,p}$ can be fetched from the L2 cache fast enough, it is better to place it there since there is an advantage to making m_c and k_c larger and the L2 cache is larger than the (somewhat small) L1 cache.

- Our analysis in Section B.3.2 also suggests that k_c be chosen to be large. Since the submatrix $B_{p,j}$ is reused multiple times for many iterations of the "fourth loop around the micro-kernel" it mays to choose n_c so that $k_c \times n_c$ submatrix $B_{p,j}$ stays in the L3 cache.

- In this scheme, the $m_r \times n_r$ submatrices of $C_{i,j}$ end up in registers and the $k_c \times n_r$ "micro-panel" of $B_{p,j}$ ends up in the L1 cache.

Data in the L1 cache typically is also in the L2 and L3 caches. The update of the $m_r \times n_r$ submatrix of C also brings that in to the various caches. Thus, the described situation is similar to what we described in Section B.3.2, where "the cache" discussed there corresponds to the L2 cache here. As our discussion proceeds, we will try to make this all more precise.

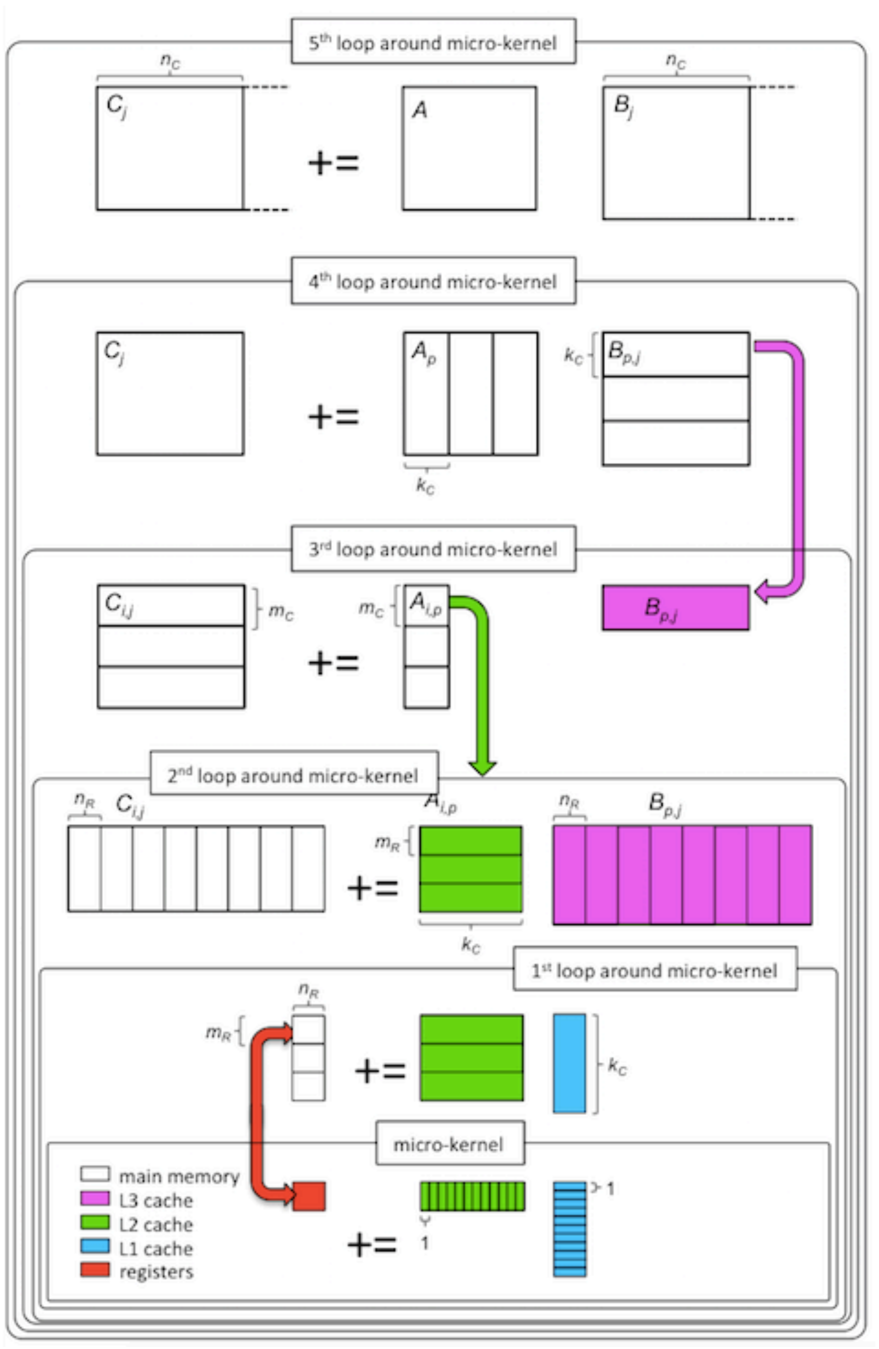


Figure B.9: Blocking for multiple levels of cache.

Homework B.3.3.2 Modify the code in Figure B.8, so that is captures the algorithm in Figure B.9.

- Copy the file `GemmIJP_JI_Kernel_MRxNR.c` into file `GemmXYZ_JI_Kernel_MRxNR.c` and modifying it appropriately, choosing `XYZ` to reflect the chosen loop ordering;
- Modify the `Makefile` appropriately.
- Execute!

You may want to pick a microkernel with a favorable $m_r \times n_r$. Try o You will want to play with the blocking sizes `MC`, `NC`, and `KC` in `GemmXYZ_JI_Kernel_MRxNR.c`.

☛ SEE ANSWER

☛ DO EXERCISE ON edX

B.4 Further Optimizing the Micro-kernel

B.4.1 Packing

The next step towards optimizing the micro-kernel recognizes that computing with contiguous data (accessing data with a "stride one" access pattern) improves performance. In our prototypical example where the micro-tile is $m_r \times n_r = 4 \times 4$, the fact that this micro-tile is not in contiguous memory is not particularly important. The cost of bringing it into the vector registers from some layer in the memory is mostly inconsequential. It is the repeated accessing of the elements of A and B that can benefit from stride one access.

Two successive rank-1 updates of the micro-tile can be given by

$$\begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\ \gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{0,3} \\ \gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{0,3} \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{0,3} \end{pmatrix} +:=$$

$$\begin{pmatrix} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{pmatrix} \begin{pmatrix} \beta_{p,0} & \beta_{p,1} & \beta_{p,2} & \beta_{p,3} \end{pmatrix} + \begin{pmatrix} \alpha_{0,p+1} \\ \alpha_{1,p+1} \\ \alpha_{2,p+1} \\ \alpha_{3,p+1} \end{pmatrix} \begin{pmatrix} \beta_{p+1,0} & \beta_{p+1,1} & \beta_{p+1,2} & \beta_{p+1,3} \end{pmatrix}.$$

Since A and B are stored with column-major order, the four elements of $\alpha_{[0:3],p}$ are contiguous in memory, but they are (generally) not contiguous with $\alpha_{[0:3],p+1}$. Elements $\beta_{p,[0:3]}$ are (generally) also not contiguous. The access pattern during the computation by the micro-kernel would be

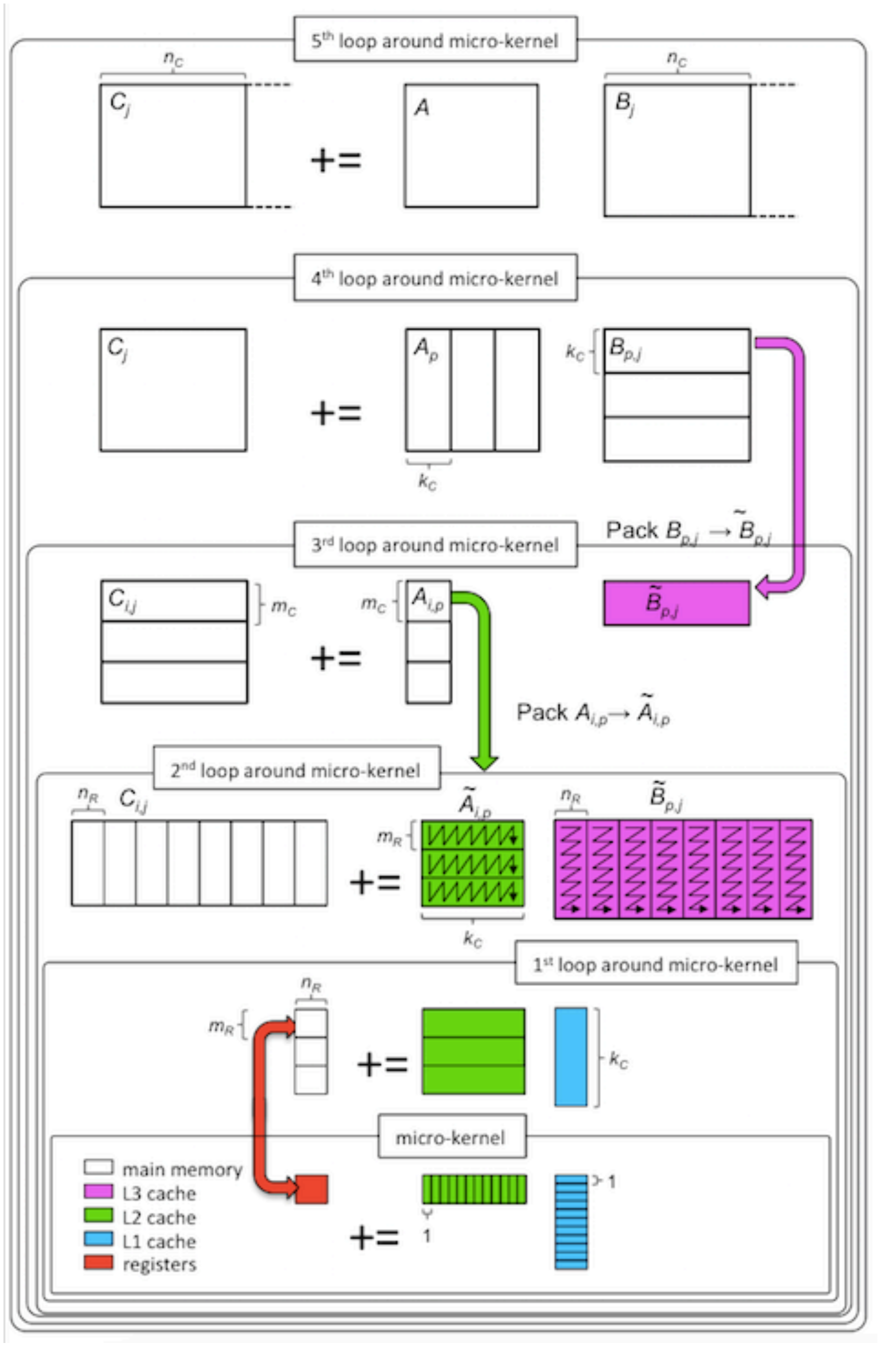

5th loop around micro-kernel
n_C
C_j
+=
A
B_j
4th loop around micro-kernel
C_j
A_p
k_C
$B_{p,j}$
Pack $B_{p,j} \rightarrow \tilde{B}_{p,j}$
3rd loop around micro-kernel
$C_{i,j}$
m_C
$A_{i,p}$
$\tilde{B}_{p,j}$
Pack $A_{i,p} \rightarrow \tilde{A}_{i,p}$
2nd loop around micro-kernel
n_R
$\tilde{A}_{i,p}$
m_R
1st loop around micro-kernel
micro-kernel
main memory
L3 cache
L2 cache
L1 cache
registers
1

Figure B.10: Blocking for multiple levels of cache, with packing.

```c
void LoopFive( int m, int n, int k, double *A, int ldA,
       double *B, int ldB, double *C, int ldC )
{
  double *Atilde = ( double * ) malloc( MC * KC * sizeof( double ) );
  double *Btilde = ( double * ) malloc( KC * NC * sizeof( double ) );

  for ( int j=0; j<n; j+=NC ) {
    int jb = min( NC, n-j );    /* Last loop may not involve a full block */
    LoopFour( m, jb, k, A, ldA, Atilde &beta( 0,j ), ldB, Btilde, &gamma( 0,j ), ldC );
  }
}

void LoopFour( int m, int n, int k, double *A, int ldA, double *Atilde,
         double *B, int ldB, double *Btilde, double *C, int ldC )
{
  for ( int p=0; p<k; p+=KC ) {
    int pb = min( KC, k-p );    /* Last loop may not involve a full block */
    PackPanelB_KCxNC( pb, n, &beta( p, 0 ), ldB, Btilde );
    LoopThree( m, n, pb, &alpha( 0, p ), ldA, Btilde, C, ldC );
  }
}

void LoopThree( int m, int n, int k, double *A, int ldA, double *Atilde,
    double *Btilde, double *C, int ldC )
{
  for ( int i=0; i<m; i+=MC ) {
    int ib = min( MC, m-i );    /* Last loop may not involve a full block */
    PackBlockA_MCxKC( ib, k, &alpha( i, 0 ), ldA, Atilde );

    LoopTwo( ib, n, k, Atilde, Btilde, &gamma( i,0 ), ldC );
  }
}

void LoopTwo( int m, int n, int k, double *Atilde, double *Btilde, double *C, int ldC )
{
  for ( int j=0; j<n; j+=NR ) {
    int jb = min( NR, n-j );
    LoopOne( m, jb, k, Atilde, &Btilde[ j*k ], &gamma( 0,j ), ldC );
  }
}

void LoopOne( int m, int n, int k, double *Atilde, double *MicroPanelB, double *C, int ldC )
{
  for ( int i=0; i<m; i+=MR ) {
    int ib = min( MR, m-i );
    MicroKernel_MRxNR( ib, n, k, &Atilde[ i*k ], MicroPanelB, &gamma( i,0 ), ldC );
  }
}
```

github/LAFF–On–PfHPC/Assignments/WeekB/C/GemmFiveLoops.c

Figure B.11: Blocking for multiple levels of cache, with packing.

```
void PackMicroPanelA_MRxKC( int m, int k, double *A, int ldA, double *Atilde )
/* Pack a micro-panel of A into buffer pointed to by Atilde.
   This is an unoptimized implementation for general MR and KC. */
{
  /* March through A in column-major order, packing into Atilde as we go. */

  if ( m == MR )   /* Full row size micro-panel.*/
    for ( int p=0; p<k; p++ )
      for ( int i=0; i<MR; i++ )
        *Atilde++ = alpha( i, p );
  else /* Not a full row size micro-panel.  We pad with zeroes. */
    for ( int p=0; p<k; p++ ) {
      for ( int i=0; i<m; i++ )
        *Atilde++ = alpha( i, p );
      for ( int i=m; i<MR; i++ )
        *Atilde++ = 0.0;
    }
}

void PackBlockA_MCxKC( int m, int k, double *A, int ldA, double *Atilde )
/* Pack a MC x KC block of A.  MC is assumed to be a multiple of MR.  The block is packed
   into Atilde a micro-panel at a time. If necessary, the micro-panel is padded with rows
   of zeroes. */
{
  for ( int i=0; i<m; i+=MR ){
    int ib = min( MR, m-i );

    PackMicroPanelA_MRxKC( ib, k, &alpha( i, 0 ), ldA, Atilde );
    Atilde += MR * k;
  }
}
```

github/LAFF–On–PfHPC/Assignments/WeekB/C/Pack.c

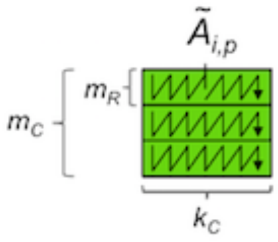
Figure B.12: A reference implementation of routines for packing $A_{i,p}$.

```
void PackMicroPanelB_KCxNR( int k, int n, double *B, int ldB, double *Btilde )
/* Pack a micro-panel of B into buffer pointed to by Btilde.
   This is an unoptimized implementation for general KC and NR. */
{
  /* March through B in row-major order, packing into Btilde as we go. */

  if ( n == NR ) /* Full column width micro-panel.*/
    for ( int p=0; p<k; p++ )
      for ( int j=0; j<NR; j++ )
        *Btilde++ = beta( p, j );
  else /* Not a full row size micro-panel.  We pad with zeroes. */
    for ( int p=0; p<k; p++ ) {
      for ( int j=0; j<n; j++ )
        *Btilde++ = beta( p, j );
      for ( int j=n; j<NR; j++ )
        *Btilde++ = 0.0;
    }
}

void PackPanelB_KCxNC( int k, int n, double *B, int ldB, double *Btilde )
/* Pack a KC x NC panel of B.  NC is assumed to be a multiple of NR.  The
   block is packed into Btilde a micro-panel at a time. If necessary, the
   micro-panel is padded with columns of zeroes. */
{
  for ( int j=0; j<n; j+= NR ){
    int jb = min( NR, n-j );

    PackMicroPanelB_KCxNR( k, jb, &beta( 0, j ), ldB, Btilde );
    Btilde += k * jb;
  }
}
```

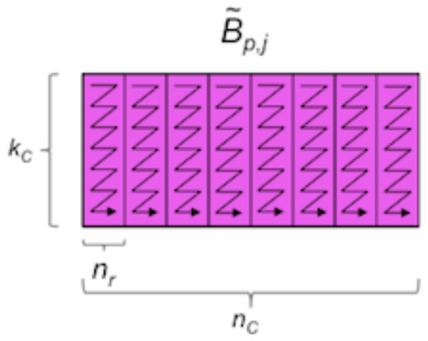
github/LAFF–On–PfHPC/Assignments/WeekB/C/Pack.c

Figure B.13: A reference implementation of routines for packing $B_{p,j}$.

much more favorable if the A involved in the microkernel was packed in column-major order with leading dimension m_r:



and B was packed in row-major order with leading dimension n_r:



If this packing were performed at a strategic point in the computation, so that the packing is amortized over many computations, then a benefit might result. These observations are captured in Figure B.10 and translated into an implementation in Figure B.11. Reference implementations of packing routines can be found in Figures B.12 and B.13. The (very modest) performance benefit is illustrated in Figure B.5 (bottom-left). While these implementations can be optimized, the fact is that the cost when packing is in the data movement between main memory and faster memory. As a result, optimizing the packing has very little effect.

Homework B.4.1.3

- Copy the file `GemmFiveLoops_4x4.c` into file `GemmFiveLoops_??x??.c` where `??x??` reflects the $m_r \times n_r$ kernel you developed before. Modify that file to use your kernel;
- Modify the `Makefile` appropriately.
- Execute!

☛ SEE ANSWER
☛ DO EXERCISE ON edX

B.4.2 Further optimizations to try.

We now give a laundry list of further optimizations that can be tried.

Different $m_r \times n_r$ choices As mentioned, there are many choices for $m_r \times n_r$. On the Haswell architecture, it has been suggested that 12×4 or 4×12 are good choices. The choice 4×12 where elements of B are loaded into vector registers and elements of A are broadcast has some advantage.

Prefetching. Elements of $A_{i,p}$ in theory remain in the L2 cache. Loading them into registers may be faster if they are prefetched into the L1 cache, overlapping that movement with computation. There are intrinsic instructions for that. Some choices of m_r and n_r lend themselves better to exploiting this.

Loop unrolling. There is some overhead that comes from having a loop in the micro-kernel. That overhead is in part due to loop indexing and branching overhead. The loop also can get in the way of the explicitly reordering of operations towards some benefit. So, one could turn the loop into a long sequence of instructions. A compromise is to "unroll" it more modestly, decreasing the number of iterations while increasing the number of operations performed in each iteration.

Different m_c, n_c and/or k_c choices. There may be a benefit to playing around with the various blocking parameters.

Using in-lined assembly code Even more control over where what happens from using in-lined assembly code. This will allow prefetching and how registers are used to be made more explicit. (The compiler often changes the order of the operations with intrinsics are used in C.)

Placing a block of B in the L2 cache. There is a symmetric way of blocking the various matrices that then places a block of B in the L2 cache. This has the effect of accessing C by rows, if C were accessed by columns before, and vise versa. For some choices of $m_r \times n_r$, this may show a benefit.

Repeat for single precision!

Repeat for another architecture!

Repeat for other MMM-like operations!

B.5 Enrichment

B.6 Wrapup