Week B

Further Optimization

B.1 Ladies and Gentlemen, Start Your Engines!

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

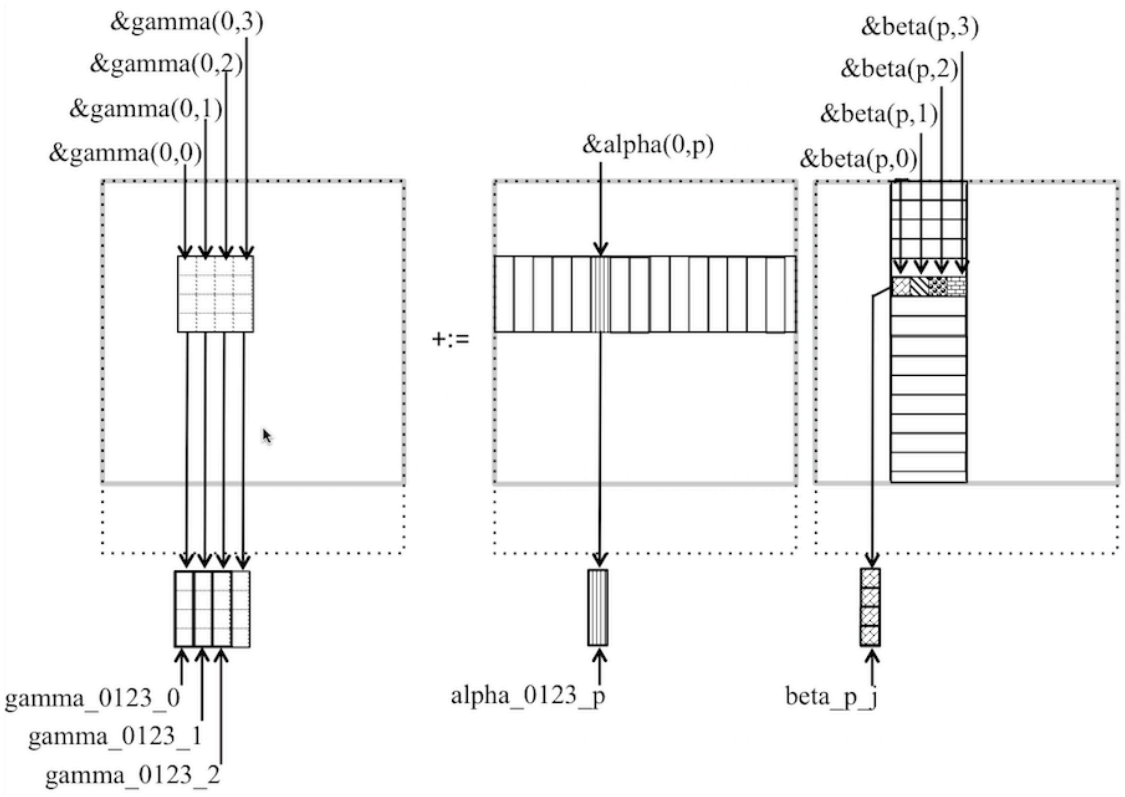

Figure B.1: Illustration of how register variables are used to implement the update of a 4×4 submatrix of C. (Repeat of Figure **??**.)

B.2 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 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

```
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.)

be used for elements of each of the matrices?

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

B.2.1 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 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 under the constraint that m_r must be a multiple of 4.

Homework B.2.1.1 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.2.2 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.2.2.1 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 `WeekB/C/GemmJI_Kernel_MRxNR.c` and can be tested with Live Script `WeekB/C/TestGemmJI_Kernel_MRxNR.mlx`. Modify it to implement your choice of kernel (meaning you can pick m_r and n_r. **Note:** you need to modify the Makefile so that `NFIRST` and `NINC` are multiples of m_r and n_r, since the implementation inherently only works for those cases.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

B.2.3 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.

```
#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_m4xn4
        ( 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 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.

B.3 Cache Memory

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 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

```
#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 ) );

  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 ) */
    __m256d 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 ) */
    __m256d 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 ) */
    __m256d 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 ) */
    __m256d 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 $m_r = n_r = 4$.

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 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.5. 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 32Kbyte level-1 cache (L1 cache) that can thus store 4K floats. Somewhat further it has a ??Mbyte level-2 cache (L2 cache) that can hold ?? floats. Further away yet, but still on chip, it has a ?? Mbyte level-3 cache (L3 cache) that can hold ?? floats.

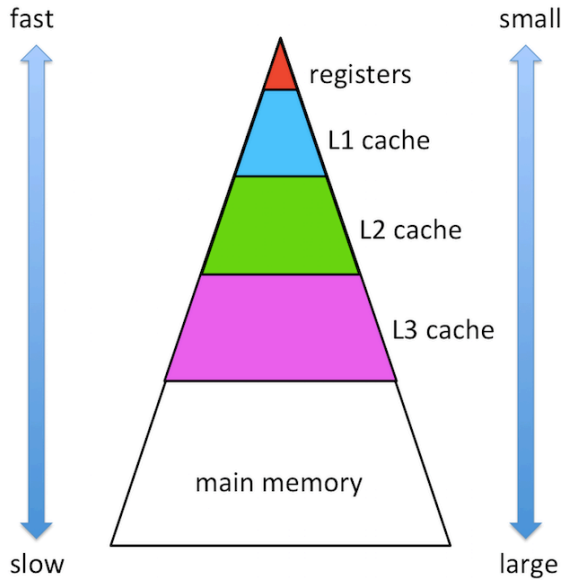

Figure B.5: Illustration of the memory hierarchy.

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 A first solution

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.

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

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

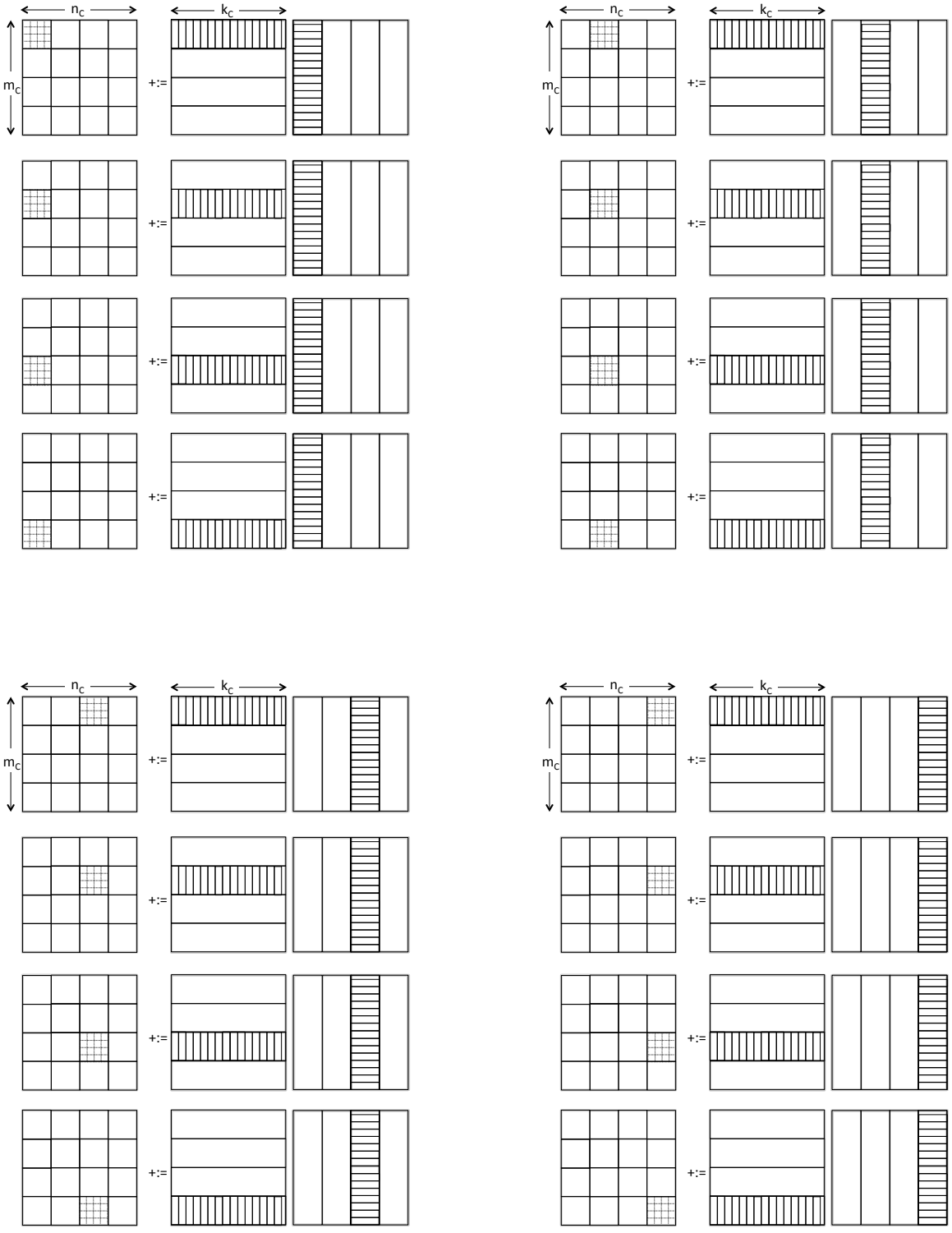

Figure B.6: Illustration of computation in the cache with one block from each of A, B, and C.

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

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

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. We illustrate the computation with one set of three submatrices (one per matrix A, B, and C) in Figure B.6 for the case where $m_r = n_r = 4$ and $m_C = n_C = k_C = 4m_r$.

B.4 Enrichment

B.5 Wrapup