software.intel.com

Putting Your Data and Code in Order: Optimization and Memory – Part 1

14-17 minutos

This series of two articles discusses how data and memory layout affect performance and suggests specific steps to improve software performance. The basic steps shown in these two articles can yield significant performance gains. Many articles on software performance optimization address parallelism as either: distributed memory parallelism (such as MPI), shared memory parallelism (such as threads) or SIMD (also called vectorization), but true parallelism should address all three areas. While these items are very important, memory is equally important and is often overlooked. Changes to software architecture and parallel design impact memory and performance.

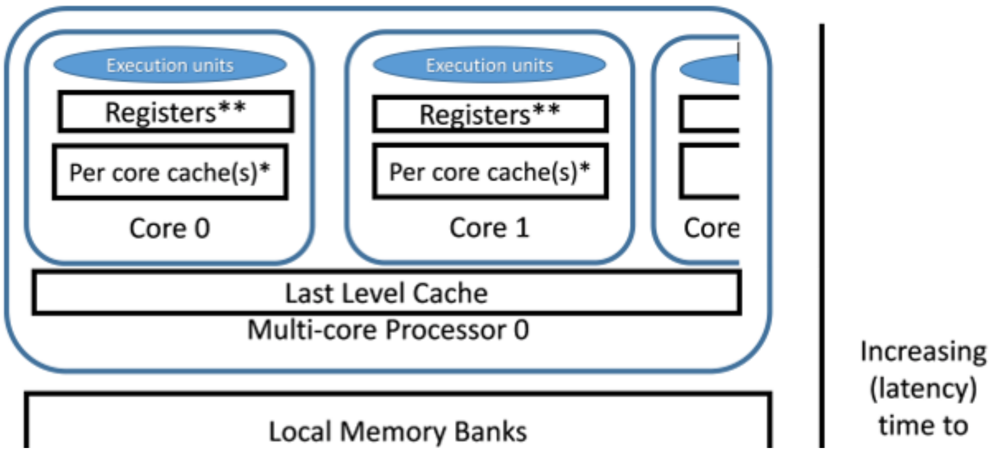
These two articles are designed at an intermediate level. It is assumed the reader desires to optimize software performance using common C, C++ and Fortran* programming options. The use of assembly and intrinsics is left to the advanced users. The author recommends those seeking advanced material to familiarize themselves with processor instruction set architecture and check out the numerous research journals for excellent articles for analyzing and designing data positioning and layout.

Putting your data and code in order builds on two basic principles: minimize data movement and place data close to where it is used. When data is brought into or close to the processor register use it as much as it is may be needed before it is evicted or moved further away from the processor execution units.

Data Positioning

Let's consider the different levels where data may reside. The closest point to execution units are the processor registers. Data in the registers may be acted upon immediately; incremented, multiplied, added, used in a comparison or boolean operation. Each core in a multicore processor typically has a private first level cache (called L1). Data can be moved from the first level cache to a register very quickly. There may be several levels of cache, at minimum the last level of cache (called LLC) is typically shared among all cores in the processor. Intermediate levels of cache vary depending on the processor whether they are shared or private. On Intel platforms, the caches maintain coherency across a single platform even when there are multiple sockets. Data movement from the caches to the registers is faster than data fetches from main memory.

A figurative illustration of data positioning, proximity to the processor registers and relative time to access is illustrated in Figure 1. The closer a block is to the register the faster the movement or the lower the latency in getting data into the register for execution. Caches are the fastest or lowest latency. Main memory is the next fastest. There may be multiple levels of memory, although different levels of memory are not addressed until Part 2. When pages are swapped out on to a hard disk or SSD, virtual memory may be significantly slower. The traditional MPI send/receive over an interconnect fabric (ethernet, infiniband, or other) has higher latency than getting data locally. Data movement from a remote system accessed through something like MPI varies in its access time depending on the interconnect fabric – ethernet, infiniband, Intel® True Scale, or Intel® Omni Scale.



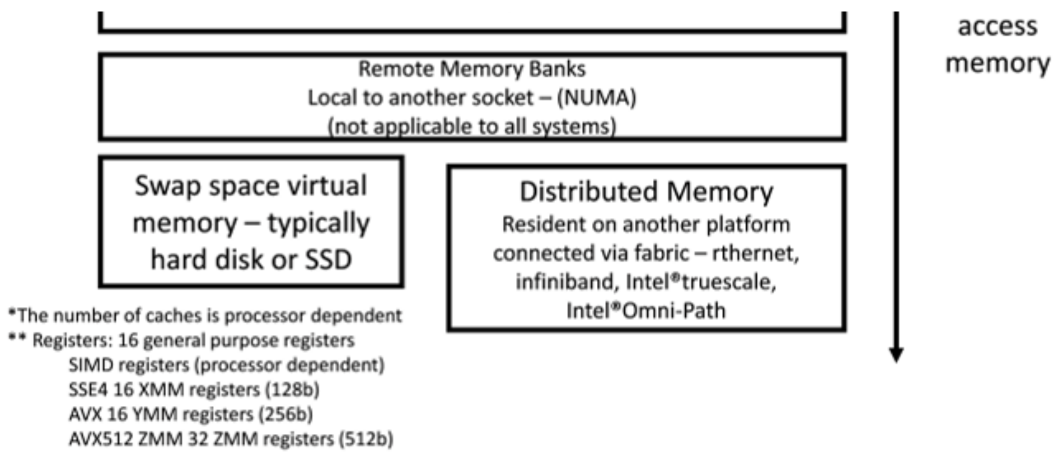

access
memory
Remote Memory Banks
Local to another socket – (NUMA)
(not applicable to all systems)
Swap space virtual memory – typically hard disk or SSD
Distributed Memory
Resident on another platform connected via fabric – rthernet, infiniband, Intel®truescale, Intel®Omni-Path
*The number of caches is processor dependent
** Registers: 16 general purpose registers
SIMD registers (processor dependent)
SSE4 16 XMM registers (128b)
AVX 16 YMM registers (256b)
AVX512 ZMM 32 ZMM registers (512b)

Figure 1. Latency memory access, showing relative time to access data.

The closest points to the execution units are the processor registers. Given the number of registers, the latency of time it takes to load data into the registers, and the width of the memory operations queue, it isn't possible to use each value in the registers only once and bring in data fast enough to keep all of the execution units fully busy. Once data is close to the execution unit it is desirable to increase its reuse before it is evicted from cache or removed from a register. In fact, some variables exist only as variable registers and are never stored to main memory. The compiler does an excellent job of recognizing when a variable scope is best as a register only variable, so using the "`register`" keyword in C/C++ is not recommended. Compilers already do a good job of recognizing this optimization and the compiler is permitted to ignore the "`register`" keyword.

The main point is for the software developer to look at the code, think about how data is used, how long it needs to exist. As yourself: Should I create a temporary variable? Should I create a temporary array? Do I need to keep this many temporary variables around? When going through a performance improvement process a developer should begin by collecting software performance metrics and focusing their efforts on data locality to the modules or branches of the code where significant portions of time is spent executing the code. A few of the popular performance data collection utilities include Intel® VTune™ Amplifier XE, gprof, and Tau*.

Data Use and Reuse

An excellent example for teaching these two steps is to consider matrix multiply. A matrix multiply A = A + B*C for three square n-by-n matrices can be represented by three simple nested `for` loops as shown below:

```
4          A[i][j] += B[i][k]* C[k][j] ;
```

The main problem with this ordering is that it contains a reduction operation (lines 138 and 139). The left hand side of line 139 is a single value. Although the compiler will partially unroll the loop at line 138 to fill the width of the SIMD registers and form 4 or 8 products from elements of B and C, those products must be summed into a single value. Summing up 4 or 8 values into one position is a reduction operation which doesn't provide the same parallel performance or utilize the full width of the SIMD units effectively. Minimizing or eliminating the use of reduction operations improves parallel performance. Seeing a single value on the left hand side inside a loop is an indication of a probable reduction. The data access pattern for a single iteration of line 137 is shown below in Figure 2 (`i,j=2`).

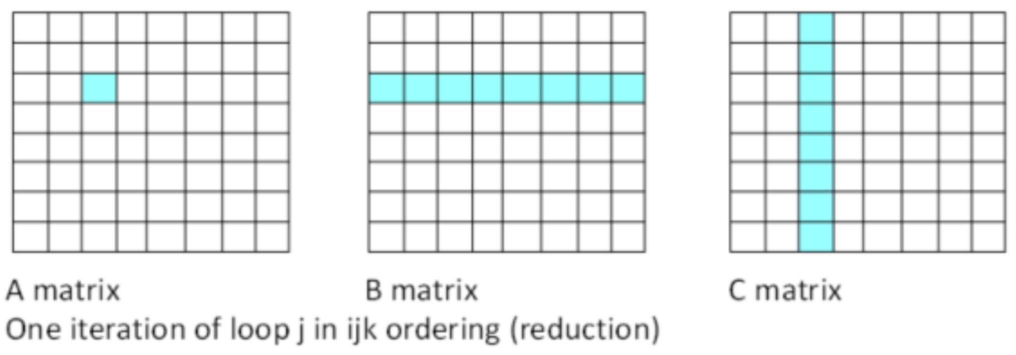


Figure 2. Ordering; showing a single value in matrix A.

Sometimes the reduction can be removed by reordering operations. Consider the ordering where the two inner loops are swapped. The number of floating point operations is the same. However, because the reduction operation or summing of values into a single left-hand side is eliminated, the processor can utilize the full width of the SIMD execution units and registers every time. This improves performance significantly.

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

When this is done, elements of A and C are accessed contiguously.

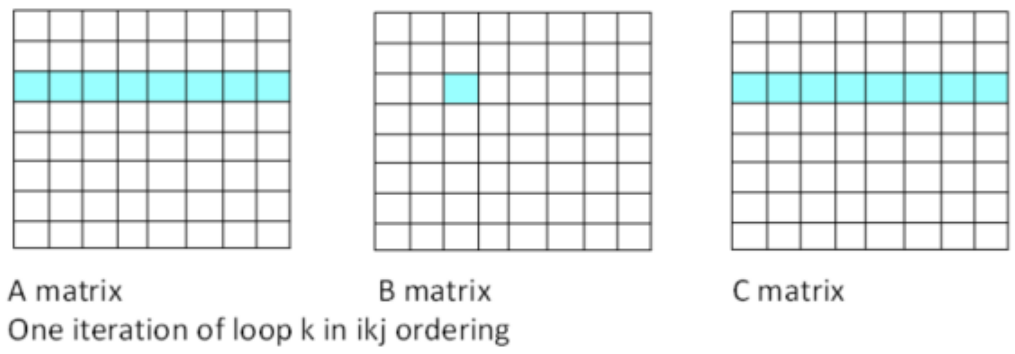


Figure 3. Updated ordering, showing contiguous access.

The original `ijk` ordering is a dot product method. A dot product of two vectors is used to calculate the value of each element of A. The `ikj` ordering is a saxpy or daxpy operation. A multiple of one vector is added to another vector. Both dot products and axpy operations are level 1 BLAS routines. In the `ikj` ordering there is no reduction required. A subset of the row of C is multiplied by a scalar from matrix B and added to a subset of the row of A (the compiler will determine the subset size depending on the width of the SIMD registers targeted – SSE4, AVX or AVX512). Memory access for a single iteration of loop `137new` is shown in Figure 3 above (again `i,j`=2).

The elimination of the reduction in the dot product view is a significant improvement. With optimization level O2, both the Intel compiler and gcc* generate vectorized code utilizing the SIMD registers and execution units. In addition, the Intel compiler automatically swaps the order of the `j` and k loops. This is revealed by looking at the compiler optimization report, which is available using the opt-report compiler option (`-qopt-report` in Linux*). The optimization report is placed (by default) into a file named `filename.optrpt`. For this case the optimization report contains the following text excerpts:

```
LOOP BEGIN at mm.c(136,4)
```

```
   remark #25444: Loopnest Interchanged: ( 1 2 3 ) --> ( 1 3 2 )
```

The report also shows that the interchanged loop was vectorized:

```
LOOP BEGIN at mm.c(137,7)
    remark #15301: PERMUTED LOOP WAS VECTORIZED
```

The gcc compiler (version 4.1.2-55) does not automatically make this loop swap. The developer will need to make this swap.

Additional performance gains come from blocking the loops to improve data reuse. In the above representations (Figure 3) for each iteration of a middle loop two vectors of length n are referenced (as well as a scalar), and each element of those two vectors is only used once! For large n it is quite common for each element of the vector to be evicted from caches between each middle loop iteration. When the loops are blocked to provide some data reuse performance jumps again.

The final code shows the j and k loops reversed and the addition of blocking. The code operates on submatrices or blocks of the matrix at a time, of size `blockSize`, in this simple case `blockSize` is a multiple of n the code.

```
for (i = 0; i < n; i+=blockSize)
   for (k=0; k<n ; k+= blockSize)
      for (j = 0 ; j < n; j+=blockSize)
         for (iInner = i; iInner<j+blockSize; iInner++)
            for (kInner = k ; kInner<k+blockSize; kInner++)
               for (jInner = j ; jInner<j+blockSize ; jInner++)
                 a[iInner,jInner] += b[iInner,kInner] *
```

In this model the data access for one particular iteration of

loop j may look like this:

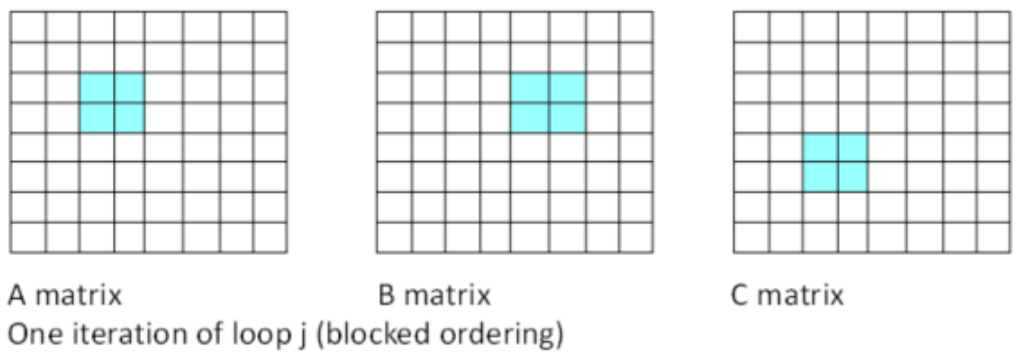


Figure 4. block model representation.

If the blocksize is selected appropriately, it is safe to assume that each block will remain in cache during the operations of the inner three loops (and maybe even in the SIMD registers). Each value or element of A, B, and C will be used `blockSize` times before it is removed from the SIMD registers or evicted from cache. This increases data reuse by a factor of `blockSize` times. When a modest size matrix is used, there is little to no gain from blocking. As the matrix size increases the performance difference is significant.

The table below shows some performance ratios measured on a system using different compilers. Please note the Intel compiler automatically swaps loops at lines `137` and line `138`. So the Intel compiler doesn't show significant differences between `ijk` and `ikj` orderings. This also makes the baseline performance for the Intel compiler much better, so the final speedup over baseline appears lower – this is because the Intel baseline performance is already so fast.

Ordering	Matrix/block size	Gcc* 4.1.2 -O2 speed up/ performance ratio over base	Intel compiler 16.1 -O2 speed up/ performance ratio over base
ijk	1600	1.0 (base)	12.32
ikj	1600	6.25	12.33

Ordering	Matrix/block size	Gcc* 4.1.2 -O2 speed up/ performance ratio over base	Intel compiler 16.1 -O2 speed up/ performance ratio over base
ikj block	1600/8	6.44	8.44
ijk	4000	1.0 (base)	6.39
ikj	4000	6.04	6.38
ikj block	4000/8	8.42	10.53

Table 1. Performance ratios measured on gcc* and the Intel compiler.

The code sample here is not complicated and both compilers will generate simd instructions. This is an older gcc compiler, it isn't meant to compare compilers, it serves an important teaching principal - it helps to illustrate the impact of order of operations and impact of reductions even when the data going into the reduction can be done in parallel. Many loops are more complicated and no compiler will recognize all the opportunities. So it can be worthwhile for developers to consider the time intensive regions of code, check compiler reports to see if the compiler has already done it or consider doing it themselves. Second notice that the importance of blocking data is when the data becomes large. There is no performance increase for the smaller of the two matrices. For the larger matrices the performance gain is significant. So before blocking all of their code developers should consider relative data size and cache. With the addition of a few nested loops and appropriate bounds developers can achieve performance gains that are 2 to 10 times faster than their original code sections. This is a big performance boost for only a reasonable amount of effort.

Using Optimized Libraries

There is still more work to be done for the truly motivated. The blocked code above is still far behind the performance improvement that would be delivered by using the level 3 BLAS routines DGEMM from one of the optimized LAPACK libraries such as Intel® Math Kernel Library (Intel® MKL). For common linear algebra and Fourier transforms, modern libraries such as Intel Math Kernel Library offer even better optimizations than what will be achieved by simple blocking and reordering exercises covered in this paper. When available, the developer should utilize these optimized libraries that are performance tuned.

Even though optimized libraries exist for matrix multiply, optimized libraries do not exist for all the possible situations where blocking improves performance. The matrix multiply provides an easy to visualize example to teach the principle. This works very well for finite difference stencils as well.

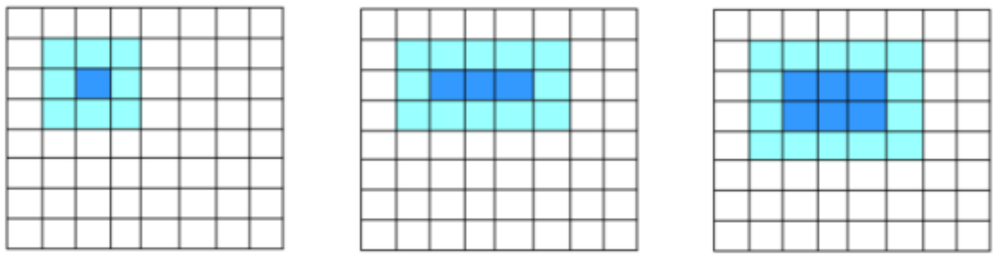


Figure 5. 2d block model representation.

A simple nine point stencil would use the highlighted blocks below to update the values in the center block. Nine values are brought in to update one position. When its neighbor is updated, six of those values will be used again. If the code progresses in an order through the domain as shown behavior will be like that shown in the adjacent figure – then we can see that to update three positions, 15 values must be brought in. Asymptotically this approaches a 1:3 ratio.

If we put things into 2d blocks again then as shown in Figure 5, six positions are updated with only 20 values being brought into the registers, for a block width of two; this would asymptotically approach a 1:2 ratio.

I encourage readers interested in finite difference techniques to study the excellent article: Eight Optimizations for 3-Dimensional Finite Difference (3DFD) Code with an Isotropic (ISO) by Cedric

Andreolli. In addition to blocking, this paper also addresses other memory optimization techniques discussed in these two articles.

Summary

In summary, there are three key steps illustrated in this paper that developers can apply to their software. First order operations to avoid parallel reductions. Second look for data reuse opportunities and restructure nested loops into blocks to allow for data reuse. This can double performance of some operations. Lastly, use optimized libraries when available; they are much faster than typical developers will obtain with simple reorderings.

Download the Code Sample

In Part 2, I will consider parallelism across multiple cores, not just SIMD registers and will address topics of false sharing and arrays of structures.