Black Belt Itanium® Processor Performance: Data Blocking and Multi Level Cache Usage (Part 3 of 5)

**By**[**David Levinthal**](https://int4-software.intel.com/en-us/user/335758)**, published on March 5, 2012**

**by David Levinthal**

**Introduction**

This paper, third in a series of articles on optimizing Itanium® processor family software, focuses on graphics-intensive and other High Performance Computing applications. The future articles will cover the use of opcode matching and data ear events in performance analysis, visual inspection of assembler in the Intel® VTune™ analyzer and a variety of other topics.

* [The Foundation](https://int4-software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-the-foundation-part-1-of-5)
* [Performance Monitoring Capabilities](https://int4-software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-performance-monitoring-capabilities-part-2-of-5)
* [Data Blocking and Multi Level Cache Usage](https://int4-software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-data-blocking-and-multi-level-cache-usage-part-3-of-5)
* [Use of Constrained Event Collection](https://int4-software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-use-of-constrained-event-collection-part-4-of-5)
* [Visual Inspection of Compiler-Generated Assembly](https://int4-software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-visual-inspection-of-compiler-generated-assembly-part-5-of-5)

Use of a multi-level cache system should not be left to chance. If an application is to take effective advantage of the secondary or tertiary cache memories then that usage must be designed into the algorithm and the data organization and access patterns. Compilers may in some cases be able to recognize data blocking opportunities but the algorithm and loop structure modifications are usually best done manually at the source level as data organization modifications are frequently required. This document will discuss the underlying performance gains that can be garnered and what execution inefficiencies are addressed when doing so.

**Caches and Latency**

A compiler will usually take into account the first or primary level cache latency and size in its scheduling algorithm. The prefetch latencies to main memory will also be included for processors like those of the Itanium processor family, which don't rely on a hardware prefetch engine. The primary cache might be a larger second level if the added latency can be absorbed in the scheduling, but typically only one level of cache is heavily used by the resulting instruction stream. The higher level caches become simply a part of the data delivery pipeline by which data gets delivered to the processor for consumption The benefits of the use of a high level cache mainly arise from the data set being small enough for a reasonable fraction of it to reside in that cache during the execution.

Only the lowest level caches access data element by element. In the case of Itanium® processors for example, the L1 cache is for integer data and the L2 cache is for floating point data. Once a data request is escalated to a higher level cache, say due to a cache miss in the lower level, what is manipulated is the entire cache line. On Itanium processors, the cache line is 64 Bytes for L1 and 128 Bytes for L2 and L3. The underlying assumption in the concept of cache lines is that applications need data that are located in nearby addresses (spatial locality). A loop streaming through some arrays is an example of this case. It is extremely important for the location and access of data and data structures be aligned in memory in order to take advantage of the cache-line based hardware feature of data access.

On Itanium processors, usage of the L3 cache can easily be measured with the VTune Performance Analyzer by simply collecting data on the events called**L3\_READS.DATA\_READS.ALL** and **L3\_READS.DATA\_READS.MISS**. These events measure what their names imply, all data read accesses to the L3 cache and all the read accesses to the L3 cache that missed, respectively. The event **L3\_READ.DATA\_READ.HIT** can be collected directly but can also be computed from the difference of the previous two. In many programs the fraction of L3 reads that are hits is almost zero. Unless the programmer does something explicit in the algorithm to manipulate the data in reused blocks that are tailored to fit in L3 cache, L3 hits will only occur if the working data set just happens to all fit into L3 cache, which is 1~6MB in size. In most applications the data sets are usually much larger than this.

To illustrate, consider a series of loops that manipulate several one-dimensional arrays, with each array using 20 MB of memory, much larger than the size of the level 3 cache. The first loop streams-in the data for the first three arrays and computes some result. 60 MB of data have been brought in, so what remains in the L3 cache are the bits of the data elements near the ends of the arrays. If the second loop uses two of those arrays, the data for them will have to be brought in again from main memory, as none of the elements at the head of the arrays are present in any part of the cache hierarchy. Depending on the application, one could consider reversing the direction of the second loop to start with and reuse the tail ends of the arrays left over from the first loop.

In general, the compiler will not know the boundaries of the arrays, so it cannot be expected to schedule execution to match the manipulated data ranges to the available caches. In some cases the compiler may merge loops to reduce data movement and the bandwidth requirements that result, as in multi-array loop case described in the previous paragraph. The best approach is for the programmer to use insight and look for performance-boosting opportunities and build and implement the necessary algorithms to exploit them. To illustrate, let us consider the case of a 1024x1024 matrix multiply of double precision floating point data elements.

**Optimizing a Dense Matrix Multiply**

One way to do this would be to use the **DGEMM** function in the Intel® Math Kernel Library, but for the sake of illustration we consider doing this explicitly. This example illustrates techniques that can be generally applied into the data organization and working data access aspects of the algorithm to best take advantage of the microprocessor caches and memory bandwidth.

We chose the C language, but the example will also work in Fortran. Remember, however, that in multi-dimensional arrays C and C++ use the last index to contiguous memory locations, while Fortran uses the firs t index. So in C, the elements **A[I][J]** and **A[I][J+1]** are adjacent in memory and almost surely in adjacent locations in the same cache line. In Fortran the elements **A(I,J)** and **A(I+1,J)** are adjacent. So for a two-dimensional array, C and Fortran store data in memory as the transpose of each other, where the data is a linear string of locations.

Our example, with MAX=1024, would be encoded as:

For(I=0; I<MAX; I++){

For(J=0; J<MAX; J++){

For(K=0; K<MAX; K++){

C[I][J] = C[I][J] + A[I][K] \* B[K][J];

}

}

}

|  |
| --- |
|  |

A simple analysis would indicate there are 2 MAX3 floating point operations. The lower bound to running this calculation is based on the total number of floating point operations. As an Itanium processor can run two fma (floating point multiply add instructions) per clock cycle, which amounts to 4 floating point operations, the limit can be expressed as:

**CPU\_CYCLES > 2 MAX3 /4**

An analysis of the memory accesses would indicate there are MAX3 stores and 3 MAX3 loads. The speed of execution will be dominated by our ability to manage the memory accesses. The use of the lowest level cache (L2 for floating point data on Itanium processors) and the processor registers results in the array C being held in a register during the inner loop and only a single row of the array A is used in the inner two loops which can reside in the L2 cache easily. The array B however gets read MAX3 times but because we loop over the first index of the array, each access brings in a cache line of which we only use a single element. In other words, when we load **B[0][0]** we bring in the cache line containing **B[0][0]** to **B[0][15]**, as sixteen double precision elements fill a cache line. Looping over the index K requires array B to access a different cache each time, making the prefetching very difficult and consuming 16 times as much bandwidth as would seem necessary. As the arrays are 8 MB each, the L3 cache is not nearly large enough to contain the working set of all 24 MB of arrays A, B, and C.

First, interchange the loops and bring the optimization level to **/O3** and **/Oa (or -fno\_alias)**. This aligns the data along cache li ne boundaries and causes the compiler to schedule reasonable prefetching. The **Oa** or **fno\_alias** flags removes the pointer ambiguity between the three arrays. (However, you need to be careful that pointer aliasing is not occurring.)

If this is not done, the loads, floating point operations, and stores, for each iteration would have to be completed before the next loop iteration could start. You cannot over-estimate the importance of disambiguating pointers! So the starting point becomes:

For(I=0; I<MAX; I++){

For(K=0; K<MAX; K++){

For(J=0; J<MAX; J++){

C[I][J] = C[I][J] + A[I][K] \* B[K][J];

}

}

}

|  |
| --- |
|  |

A simple analysis done by the VTune analyzer shows that **L3\_READS.DATA\_READS.ALL** = **L3\_READS.DATA\_READS.MISS**. In other words, no data is being recovered from the L3 cache. Furthermore, we are loading the B array MAX3 times and those memory accesses dominate the timing. The solution is to unroll the outer loop to reuse the value of **B[K][J]**.

For(I=0; I<MAX; I+=8){

For(K=0; K<MAX; K++){

For(J=0; J<MAX; J++){

C[I][J] = C[I][J] + A[I][K] \* B[K][J];

C[I+1][J] = C[I+1][J] + A[I+1][K] \* B[K][J];

through

C[I+7][J] = C[I+7][J] + A[I+7][K] \* B[K][J];

}

}

}

This improves performance but the L3 cache is still just a pass through. At this point the bank conflicts that arise from the loads of A, B and C and the different rows of each array need to be addressed. As these are 1024x1024 arrays, each successive row is bank aligned. Each row takes 8096 Bytes and therefore successive rows are 256 byte aligned. As the bank structure has a 256 Byte periodicity, successive rows will start in the same banks and the accesses required by the unrolling can create bank conflicts. Furthermore, the arrays are exactly 1MB in size so the different arrays are bank or 256 byte aligned. It should be pointed out at this point that this problem was expressly formulated to create a worst case scenario, where every possible addressing conflict that could occur actually does.

The solution is twofold. First, we increase the di mension of the arrays to 1040 but still only loop over 1024 elements. This provides a padding between successive rows. By changing the leading dimension from NUM=1024 to DIM=1040 changes the alignment of successive rows from being 256 byte aligned to being only 128 byte (8 bank) aligned. This will remove the bank conflicts created by accessing adjacent rows.

The relative alignment of the three arrays can be manipulated by allocating (using malloc) a little extra space and adjusting the base addresses of the arrays. As they were created in immediate succession, the three arrays are in fact 1MB aligned. The straightforward solution is to first force all three addresses to be 256 byte aligned, and then adjust the bases from there. The optimal solution is to move one array by 4 banks or 64 bytes, and the other by 8 banks or 128 bytes. The following code would produce such an effect.

typedef double array[DIM];

char \*buf1, \*buf2, \*buf3;

int Offset\_Addr1=128, Offset\_Addr2=192, Offset\_Addr3=0;

array \*a, \*b, \*c;

//malloc arrays space

buf1 = (char \*) malloc(DIM\*DIM\*(sizeof (double))+1024);

buf1 = buf1 + 256 - ((UINT64)buf1%256) + (UINT64)Offset\_Addr1;

buf2 = (char \*) malloc(DIM\*DIM\*(sizeof (double))+1024);

buf2 = buf2 + 256 - ((UINT64)buf2%256) + (UINT64)Offset\_Addr2;

buf3 = (char \*) malloc(DIM\*DIM\*(sizeof (double))+1024);

buf3 = buf3 + 256 - ((UINT64)buf3%256) + (UINT64)Offset\_Addr3;

a = (array \*) buf1;

b = (array \*) buf2;

c = (array \*) buf3;

**Data Blocking the Matrix Multiply**

At this point, this is about all the improvement that can be done with the encoding above. The problem is that the L3 cache is still simply acting as a pass through. In order to use the L3 cache, the algorithm must be designed accordingly. In a dense matrix multiply, this is done by doing the calculation by breaking the matrices into tiles and then multiplying the tiles and summing the tile results to reconstruct the full result. Pictorially this can be seen as follows. A matrix multiply can be visualized as:

click image for larger view  
  
The problem here is that (in C) the second input array is accessed across the cache lines. This was the original problem in the first encoding that was discussed. For the time being, we ignore this issue, and it will need to be addressed for even more profound reasons.  
  
What is of immediate concern is how to restructure the algorithm so that the multi-level cache can be utilized. The approach is to break the matrices into tiles and progressively multiply the elements in the tiles. Thus:  
  
  
  
click image for larger view  
  
Is followed by:  
  
  
  
click image for larger view  
  
The approach involves creating a tile that fills a large fraction of the L3 cache and multiply it by a series of smaller tiles from the other matrix that fill less than half of L2. The problem will require that half of L2 be available for streaming-in data from L3. Pictorially this looks like:  
  
  
  
One difficulty here is that as the elements of a column of the "B" array (the one in the rightmost) are streamed into the L2 cache for access, each e cache line of data elements is also brought in. In the picture, the cache-lines are 16-element horizontal strips (8bytes \* 16 elements = 128 Bytes = 1 cache line). Even more problematic is that the successive cache lines brought in for successive elements of the column are not contiguous in virtual (or physical) address space. In the case being considered for 1024 x1024 matrices, the jump in virtual address between adjacent elements of a column are separated by 8096 Bytes. Each double takes 8 bytes, times 1024. The of virtual address of the L2 cache is 32KB (256KB/8 for the associatively of each way). So as the loop progresses through the column, the B array uses only 4 associative sets and quickly evicts the elements of the A array.  
  
This explains why a naïve data blocking may slow down an application. The intermediate data blocks must be copied onto contiguous pieces of memory or the purpose is defeated.  
  
The solution: the tiles must occupy contiguous areas in memory. The best way to accomplish this is to copy the tile of B and transpose it at the same time. This not only moves the tile into a contiguous piece of memory, but also aligns the accesses down the "column" to be along the cache line organization.  
  
Because the tiles of A are aligned with the cache lines, they are contiguous in memory and no copying is required. The solution for this is to transpose the B matrix at the beginning of the problem. The result is that the triply nested loop we started with becomes the following.

For(I=0; I<MAX; I++){

For (J=0; J<MAX; J++){

BT[J][I] = B[I][J];

}

}

For(I=0; I<MAX; I++){

For(J=0; J<MAX; J++){

For(K=0; K<MAX; K++){

C[I][J] = C[I][J] + A[I][K] \* BT[J][K];

}

}

}

The optimal unrolling strategy becomes a little different. The problem is now symmetric with regards to its accesses of A and B. This indicates that the level of unrolling in the loops over I and J should be the same. Further the value of C is accumulated in a register so it is loaded and stored only once during the entire calculation. This reduces the number of address manipulations significantly. Unrolling each of the outer loops by 4 will result in 16 combinations of I and J. Thus with 8 loads we can execute 16 fma instructions, and that is all that will be required in the kernel of the loop. A perfect kernel will have 2 fma's being executed every clock cycle.

For(I=0; I<MAX; I++){

For (J=0; J<MAX; J++){

BT[J][I] = B[I][J];

}

}

For(I=0; I<MAX; I+=4){

For(J=0; J<MAX; J+=4){

For(K=0; K<MAX; K++){

C[I][J] = C[I][J] + A[I][K] \* BT[J][K];

C[I+1][J] = C[I+1][J] + A[I+1][K] \* BT[J][K];

C[I+2][J] = C[I+2][J] + A[I+2][K] \* BT[J][K];

C[I+3][J] = C[I+3][J] + A[I+3][K] \* BT[J][K];

Through

C[I][J+3] = C[I][J+3] + A[I][K] \* BT[J+3][K];

C[I+1][J+3] = C[I+1][J+3] + A[I+1][K] \* BT[J+3][K];

C[I+2][J+3] = C[I+2][J+3] + A[I+2][K] \* BT[J+3][K];

C[I+3][J+3] = C[I+3][J+3] + A[I+3][K] \* BT[J+3][K];

}

}

}

The data blocking is now accomplished by adding two more outer loops to schedule the ordering through the I and J indices in the manner illustrated in the diagrams above. For matrices of 1024x1024 of double precision elements, this involves blocking one matrix in units of 16 rows (16 \* 1024 \* 8 = 128 Kbytes or 1/2 of the L2 cache) and the other by 256 rows (256 \* 1024 \* 8 = 2MB = 2/3 of a 3MB L3 cache).

numi = 256;

numj = 16;

for(ii = 0; ii<NUM; ii+=numi){

for(jj = 0; jj<NUM; jj+=numj){

for(I=ii; I<ii+numi-1; I+=4) {

for(J=jj; J<jj+numj-1; J+=4) {

for(k=0; k<NUM; k++) {

followed by the same 16way unrolled kernel.

**Organization and Reuse of Data**

Key learnings from this example:

* The data organization aspect of the problem and the algorithm cannot be decoupled and must be designed together.
* The organization of the data must incorporate the physical access mechanism, to ensure that all the data that is moved in cache lines is used as completely as possible to take the maximum advantage of the memory bandwidth.
* If there is high reuse of data and the working set is much larger than the largest cache si ze then a data blocking strategy must be used. A compiler is unlikely to succeed doing this automatically.
* In a data blocking algorithm, the blocks of data must be contiguous to ensure that the entire contents of the cache can be utilized. This will almost always require that a local copy of the data blocks be made during the algorithm.

In the example above, a single reorganization (the transpose) was possible, but this is not usually the case. A related issue shows up in sparse matrix solvers. The NAS benchmark CG (conjugate gradient) illustrates a coding structure characteristic of these solvers. Here a packed array derived from the large sparse matrix is manipulated in the course of the solving process. It is indirectly addressed through an address array. An initial version is created and then a sum is computed which is used to update the packed array. The size of the packed array is determined such that it easily fits in the L3 cache. A typical encoding is as follows:

For(j=0; j<numrows; j++){

sum = 0;

for(k=rowstrt(j); k< rowstrt(j+1); k++){

sum += a(k)\*p(address(k));

}

q(j) = sum;

}

Then the packed array *p* is updated on the basis of the array *q* and the process iterates until the convergence criteria are met.

In such a problem the arrays "***a***" and "***address***" are normally much larger than the size of the L3 cache and are thus streamed through the cache structure. The array *p* is read many times for each time it is written. However the elements are accessed essentially randomly so very little use is made of the cache line organization of memory storage in the cache hierarchy. Each access of *p* will require a different cache line than the previous access of *p*. The benefit of having cache lines comes from those situations where a required cache line just happens to still be in the L2 cache. The probability of this is based on the fraction of the entire p array that can be held in L2. The arrays *a* and *address* only need to occupy a few cache lines at any given time.

The issue that arises in such a case is that the cache lines that contain the array *p* are never written to memory. Consequently they remain in a "modified" state at all times, due to having been written between iterations of the nested loops as shown above. When a cache line in a "modified" state is evicted from the L2 cache, it must be written back to the L3 cache to ensure that the caches are coherent.

The effect of this is that there is an artificial increase in the bandwidth requirements between the L3 and L2 caches due to having to transfer the cache lines back to L3 when they are only being read in the principal loop. The solution to this is to use the **\_\_fc** (flush cache) intrinsic to force the cache lines of *p* out of L3 and back to into memory. They are then read back into the L3 cache. When the cache lines are read back they are in a “shared” state, as opposed to a “modified” state. Consequently when the cache lines of the array *p* are evicted from L2, they do not need to be transferred back to L3 as they are not modified.

This can be accomplished with a simple function that loops over the range of the *p* array first flushing the modified cache lines and then prefetching them to bring them back in the shared state.

Void mem\_reset (char \*p, int len)

{

int i;

char \* addr;

addr = p;

for(i=0; i< len; i+=128){

\_\_fc(addr);

\_\_lfetch(hint\_nt1, addr)

addr += 128;

}

}

If the L2 to L3 bandwidth might be limiting performance can be investigated with the **L3\_WRITES.L2\_WB** counter. This counter increments whenever a cache line is written to L3 due to a writeback from L2. When the default version of CG is run, this ratio of L2 writebacks to all L2 data references (reads) is approximately 1:7. If the cache flush function is called after the *p* array is updated, this ratio drops to 1:150, and the execution speeds up by 15%.

**Conclusion**

We discussed how Itanium® processor family software can help optimize graphics-intensive and other High-performance Computing applications by careful use of its multi-level cache system. The next article in the Series discusses [Constrained Event Collection on Itanium® Processors](https://int4-software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-use-of-constrained-event-collection-part-4-of-5) (opcode matching).

**Related Links**

* [Introduction to Microarchitectural Software Optimization for Itanium Processors](https://int4-software.intel.com/sites/default/files/m/d/4/1/d/8/219348_software_optimization.pdf) (PDF 466KB)
* [Intel® C/C++ Compiler Users Guide](https://int4-software.intel.com/en-us/)
* [Itanium Processor Software Specifications](http://developer.intel.com/design/itanium/arch_spec.htm)
* [Intel® Itanium® Processor Reference Manual for Software Development and Optimization](http://developer.intel.com/design/itanium2/manuals/251110.htm)

**About the Author**

David Levinthal is a lead Itanium® Processor software performance support engineer for the VTune™ Performance Analyzer program. He joined Intel in 1995 working for the Supercomputing Systems Division and then the Microprocessor Software Labs. He has been a lead software support engineer on the Itanium Processor Family since 1997. Prior to joining Intel he was a Professor of Physics at Florida State University. He has received the DOE OJI award, the NSF PYI award and a Sloan foundation fellowship.