PARALLELIZING DENSE MATRIX-MATRIX MULTIPLICATION THROUGH BLOCKING ON INTEL SANDY BRIDGE AND INTEL XEON PHI

Kamran Nobahar
High Performance Computing - Spring 2015
Introduction
In this project I optimized the Dense Matrix-Matrix multiplication calculation by tiling the matrices and parallelizing the process. I ran the code on two different platforms: Intel Sandy Bridge processors and Intel Phi co-processor.

Dense Matrix-Matrix Multiplication
By definition multiplying two matrices $A$ and $B$ results to another matrix as:

$$(AB)_{ij} = \sum_{k=1}^{m} A_{ik}B_{kj}.$$ Implementing an algorithm which follows the definition ends with three nested loops and have the complexity of $O(n^3)$. One can tune the loops for cache access by alternating the indexes but that would not decrease the complexity. There are other serial algorithms which have better run time. The best one as of now is Coppersmith–Winograd which runs at $O(n^{2.3728639})$.

Matrix Blocking
Matrix blocking is the process of dividing a matrix into sub-matrices. For example if we divide a square matrix of size $n$ into a $q \times q$ array of blocks $A_{i,j}$ ($0 \leq i, j < q$) each sub-matrix will be of size $(n/q) \times (n/q)$. The multiplication in this layout can be done in two phases: First the blocks are treated as elements of the super-matrix and participate in the multiplication. So Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $0 \leq k < \sqrt{p}$. ($p$: number of blocks); then the result of each element is calculated based on a classic algorithm. In this way we will have $q^3$ matrix multiplications. Tiling matrices is done in this project for two reasons: Providing opportunity for parallelization and ultimate cache tuning.

The one important question that we should address is that how many blocks we should divide the matrix to or in the other words what size the blocks should have? Also if we are to preform each block calculation through an individual thread we would prefer to have the minimum number of synchronizations and communications between threads. These questions lead us to the cache-oblivious algorithms[2] and the threading efficiency techniques used in Cilk[3]. In cache-oblivious algorithms we basically want to take advantage of CPU cache without having the size of the cache (or the length of the cache lines, etc.) as an explicit parameter. For our case this can be done by blocking the matrix in a recursive fashion. On each recursion we tile the matrix into 4 blocks. Each of those blocks then is tiled to yet another 4 blocks. This process continues until the smallest block can fit into the lowermost level of our cache platform. For returning from the deepest recursion the actual matrix multiplication is taken place. Using this technique we can make sure that all the concerned data is in the cache and we minimize the number of cache misses. For each step we have 4 block multiplications.

The Matrix Layout
The way we store our matrices in the memory is critically important for our cache strategy. Normally matrices are stored in the memory in row-major or column-major order but these layouts cannot guarantee that on cache misses all irrelevant data we don't need for a block computation don't get loaded into our cache. For this reason we should make sure that the data for each block is stored
adjacent to each other and also is aligned properly in the memory. We use the famous “Z” layout for storing our matrices. Other layouts such as “U” and “X” also can be used. To achieve the best result we need to design the layout for the smallest block size we are going to use otherwise there due to index complexity the number of operations for accessing the right data will increase significantly. We can implement pre and post processing phases for our program to transform the layout of the matrix as we wish.

Platforms
The devised algorithm is tested on two platforms on the Stampede supercomputer. The first platform contains two Intel Xeon E5 – 8 Cores and the second one contains one Intel Xeon Phi Co-processor with 61 cores. One end of this project was too test the Intel MIC technology and benchmark it with matrix-matrix multiplications. The MIC card has a 512 bit vector unit and the clock speed of 1.31 Ghz. It supports 64 bit addressing and runs a Linux machine. They are basically different from GPGPUs in the sense that their cores unlike GPU cores are real and fully capable cores. They attach to the host system but they do not access to any shared memory with the host.

There two main ways to run the algorithm on the MIC card. Offloading and Native Execution. On the Native Execution one can easily connect to the card through SSH or other means and compile the code for the card and execute it on place. In the other way one can mark the desired part of their code to get offloaded automatically by the compiler to the MIC card from the host. This can be done by using appropriate pragmas. In either way there is no need to deal with any new libraries or learn new languages. On execution all the data and necessary code for execution on the MIC card is transferred over the card.

Results
On the following figures results from the benchmarks done on different conditions and with different algorithms are illustrated.
Figure 1 compares the most naïve serial algorithm with a serial implementation of the blocking algorithm and also a parallel execution of it on Intel Sandy Bridge CPUs. For the latter case 8 cores are used and also the smallest block has 64x64 elements.
Figure 2 demonstrates the speedup we get from moving the execution from Sandy Bridge to the MIC card. On the MIC card 128 threads on 60 cores executed the algorithm having the smallest block had the size of 128x128.
Figure 3 takes the transfer time happened for the MIC card into account. This time is the one spent on transferring all the data from Host to the card and transferring the result back. We can see that having this transfer time defies the purpose of using Intel Phi but we should consider that we could run the program natively on the card itself and skip the transferring phase completely.
Figure 4 shows two important timings to compare with what we have achieved. One is the timing of executing the code on two Xeon E5 processors. On this platform we have 16 cores which gives us a speedup more than what we have achieved. The takeaway would be that with enough number of state of the art cores 60 1.31 Ghz of MIC card cannot have a strong place in our algorithm performance. This of course doesn't mean that MIC card is useless in this sense but that we should find its best place to take responsibility.
There are two other important benchmarks on this figure: Matrix-Matrix multiplication using MKL on Sandy Bridge platform and also on the MIC card. MKL which is the hand optimized math library for the
Intel processors, shows a significantly better performance on 16 Sandy Bridge cores comparing to the implemented algorithm in this project but surprisingly it doesn't work better on the MIC card and the implementation in this project performs a little better in the terms of computation times.

![Figure 1](image1.png)

*Figure 1*

![Figure 2](image2.png)

*Figure 2*
Figure 3

Figure 4
Other considerations and future works

It is worth emphasizing on three other important tuning techniques which has been used in this project:

1. Vectorization: Both Xeon E5 and Xeon Phi cores are capable of vector computations. Their vector units are powerful tools which can be used to boost the speedup of the computation. The nature of Matrix-Matrix Multiplication is totally capable of vectorization. In this project I vectorized all the important parts of the code using compiler directives.

2. Data alignments: Efficiency in transferring data plays a huge role in reducing the latency of the system. The data in memory is not usually transferred to cache and CPU from arbitrary address points. For this reason it is worth to know how the bandwidth of the memory bus and align our data to memory by that to avoid waste in data transfers.

3. Avoiding context switches: Parallelism can backfire if we overdo it. Creating too many threads in the system causes the cores to switch between them constantly and by that they have to replenish their caches on each switch. This will defy all the purpose of our cache friendly algorithm. Other than that context switches are in general costly and impose a huge unnecessary overhead to the performance. For this reason we need to make sure that we do not create threads more than what all the core can handle in total.

As a road-map to achieve better performances we can mention some other tunings to do:

1. Using better serial matrix multiplication algorithms for the smallest block multiplication. Considering that at the deepest level of the recursion there are two small matrices which should be multiplied to each other we may better off use a better algorithm like Strassen's[4] for this multiplication than the basic one.

2. We avoided running our algorithm natively on the phi card because of the hurdles over maintaining necessary libraries but running the algorithm on the MIC card directly eliminates all the transfer time we have with the offloading technique.

3. The memory of the MIC card is limited. This makes us think what we should do when we have matrices which do not fit in our memory. The answer is distribution. Having the opportunity to distribute tasks symmetrically between Phi cores and Sandy Bridge cores we can spawn a big number of threads over not only one machine but as many machines as we wish.

References


