ABSTRACT

Cholesky decomposition is an important technique for solving a system of linear equations, since it is twice as fast as the LU decomposition, when applicable. Many scientific applications are required to solve such a system of equation. Thus it is very essential to have an implementation of it that solves very efficiently. In this work we try to find an efficient implementation on both General-Purpose Graphics Processing Unit(GPGPUs) using Compute Unified Device Architecture(CUDA) and on Intel processors using Thread Building Blocks(TBB). We also analyze the resultant implementations performance results.

1 Introduction

This project aims at implementing efficiently Cholesky Decomposition on GPGPUs using CUDA and Intel processors using TBB. The main tasks are to analyze the algorithm for the data and control dependences, find scopes to improvement by parallelizing the operations wherever possible. Use of spatial and temporal locality is also an important source of performance enhancements. After all this we also want to be space efficient and minimize the overheads involved in parallelization as these too can lead to better performance.

2 Cholesky Decomposition:

Cholesky Decomposition, also known as Cholesky triangle, is a decomposition of a symmetric, semi-definite matrix into a product of a lower triangular matrix and its conjugate transpose. As such it is an example of a square root of a matrix. It is used for solving a system of linear equations as follows.
Suppose A is an n x n symmetric, semi-definite matrix, and B is n-vector, X an unknown n-vector.
So to solve the system of equations
$$AX = B$$

we find out a lower triangular matrix $L$ such that

$$LL^T = A$$

$$\therefore LL^T X = B$$

which can then be solved as in LU decomposition.

3 Definitions

1. **Problem Size** is said to be $N$ if the dimensions of the input matrix is $N \times N$.

2. **Speedup** ($S_p$): It is the measure of how much a given parallel algorithm is faster than another algorithm which may be a sequential or a parallel algorithm itself. So

$$S_p = \frac{T_1}{T_p}$$

where, $T_1$ is the running time of the algorithm against which the performance of a parallel algorithm is being compared. $T_p$ is the running time of the parallel algorithm in question.

Now, speedup can be shown in two forms depending upon the algorithm being used to compare the performance against.

3. **Absolute Speedup**: When the algorithm used to compare the performance is the best known sequential algorithm, the resultant ratio of its running time to that of the parallel algorithm is called as Absolute Speedup. So $T_1$ here is the running time of the best sequential algorithm for that problem.

But most people do not have access to fastest serial machines on which they could run the sequential algorithm, also this measure also may change according to which machine is used to run the algorithm on. Also we may not know the best sequential algorithm itself. So it may not be always be the case that we can compute the absolute speedup for a given parallel algorithm.

As such a reasonable way out will be to use as reference the running time of some standard implementation, proven as efficient. We will be using the implementation given by ALGLIB, more about which will be said in later sections.

4. **Relative Speedup**: When the running time of the same parallel algorithm when run on a single processor is taken as $T_1$, the ratio obtained is called as Relative Speedup.

The Relative speedup got in such a way can be misleading, since a parallel algorithm does many things such as forking threads, synchronizing, etc that help on a parallel platform but when this algorithm is run on a single processor the many threads forked run on the same machine, leading to overheads due to context switches and others depending on how the synchronization is implemented. As such we also include in $T_1$ extra time that are unnecessary and incorrect. Thus such measures are misleading giving an exaggerated speedup.

So we instead of using the same parallel algorithm run on single processor as the reference, use the **serial algorithm**. By this we eliminate the unnecessary overheads. Also this emphasizes on gains from compiler optimization and parallelization techniques.
4 Implementations

We now begin with the details of work which is divided into two parts, first the implementation details on CUDA, followed by the same on TBB.

4.1 Part 1 : CUDA

In this part of the project we compare 3 algorithms, which are as follows:

1. Submatrix Update algorithm
2. Inner-product Subtraction algorithm.
3. Blocked Implementation : The implementation of this is available at [2].

We now briefly explain the individual algorithms so as to be able to better explain the nature of results and the intuitions used in our implementation.

4.1.1 Submatrix Update Algorithm

The algorithm iterates one time less than the dimension of the matrix with in each iteration carrying out 3 different operations, namely

1. **Square Rooting**: In this step we compute replace the $i^{th}$ diagonal element by its square root, where $i$ is the iteration variable. This step taken $O(1)$ time in sequential settings.

Fig 1 : Square rooting step in submatrix update algorithm.
2. **Normalization**: Each of the elements in the $i^{th}$ row after the diagonal element mentioned above is divided by the diagonal element. This step takes worst case $O(N)$.

![Normalization step in submatrix update algorithm.](image)

3. **Submatrix Update**: In this step rest of the elements of the upper triangle below the $i^{th}$ row are updated using the values in the $i^{th}$ row. This is $O(N^2)$ worst case.

![Submatrix-update step in submatrix update algorithm.](image)

4.1.2 **Inner-product Subtraction algorithm**

This algorithm is almost identical to the submatrix update algorithm except for the third operation, instead of submatrix update here we do **innerproduct subtraction** which is as follows. If the iteration
variable is \( i \) then each of the element in the \( i + 1^{th} \) row is subtracted by the inner-product of \( i + 1^{th} \) and \( k^{th} \) column if \( k \) is the index of the element of the \( i + 1^{th} \) row, above the \( i + 1^{th} \) row. The inner product is computed by multiplying the \( j^{th} \) row element of a column with the \( j^{th} \) row element of the other.

Fig 4: Square rooting step in inner-product subtraction algorithm.

Fig 5: Normalization step in inner-product subtraction algorithm.

Fig 6: Inner product subtraction step in inner-product subtraction algorithm.

4.1.3 Blocked CUDA Implementation

This algorithm iterates one time less than the number of block in a row ie. \( N \) divided by the block size. In each iteration it does 3 operations each corresponding to a kernel function, namely.

1. **Iterative step:** In this the a slightly modified submatix update algorithm described above is applied on the \( i^{th} \) diagonal block, where \( i \) is the iteration variable.

2. **Strip block update:** The elements of the blocks below the \( i^{th} \) diagonal block are updated using
the elements in the ith diagonal block. This is a costly operation involving computation of a dot-product with lot of synchronization needed between the threads, as a thread can start only when a particular thread is finished.

3. **Lower Right blocks update**: The elements in the block that are on the lower right of the $i^{th}$ diagonal block are updated by taking a dot product of a row and a column of particular blocks.

4.1.4 Results

1. **Comparison between the Sequential Implementation of algorithms**:
   - From the graph, it is clear that the inner-product subtraction algorithm works better than the submatrix update algorithm.
   - This is due to the costliness of the submatrix update step in that algorithm as compared to the inner-product subtraction step. The former is costly since in an iteration on an average $N^2/2$ elements are involved in it as compared to atmost $N^2/4$ involved in the latter.

![Fig 7: sequential inner product vs sequential submatrix.](image)

2. **Comparison of the Parallel implementations of the algorithms**: Two key observations come out of the graph, which are as follows:
   - Firstly, here the submatrix update algorithm perform better than the inner-product subtraction algorithm consistently.
   - This is due to the fact that the computation done to update each element of the submatrix can be done separate threads all in parallel, with no synchronization required.
On the other hand the computation of each element in the inner-product subtraction involves computing a dot product which is an O(lg K) time parallel algorithm, where K is the length of the vectors involved in the dotproduct, with lot synchronization needed.

Thus submatrix update is at an advantage over the other in the parallel environment of a gpgpu.

(b) Secondly, the blocked algorithm performs better than the other two till a certain problem size (≈500), but for problem size greater than this the other two algorithms perform better.

There are two competing factors at play over here, one is the greater complexity of the strip-block and lower right block steps and the other is the number of points in the program where kernel threads are forked.

On comparing the three algorithm on the basis second factor it is seen that since the all 3 algorithms have 3 points where kernel threads are forked per iteration, the blocked algorithm, since it iterates N/BLOCKSIZE times, has less such fork points.

This fork point can be costly because everytime threads are forked there is this overhead of forking threads plus the allocation of shared memory, copying from and to the global and shared memory. The time saved here compensates for the higher complexity of the algorithm initially.

But due to architectural specification the block size cannot go beyond a limit, thus after certain problem size the time saved by forking threads less often becomes constant while the overhead due to higher complexity keeps increasing making its performance to degrade. So after this point the other algorithms perform better than the blocked one.

**Fig 8:** Comparison of Parallel Implementations of the 3 algorithms.
4.2 Part 2 : Thread building blocks

For the experiments on the multi-core hardware, we chose the Intel Thread Building Blocks library. We compare three algorithms, the Sub-Matrix Update, Inner-product Subtraction and blocked Cholesky implementation. The hardware chosen for testing includes machines of 1,2,4,16,24 cores. The exact specification of the hardware is as follows:

- Intel Pentium 4 @2.4GHz (1 Core)
  Microsoft Windows XP SP3, 768 Mb RAM
  cpu MHz : 2394.000
  Cache size : 512 KB

- AMD Athlon(tm) 64 X2 Dual Core Processor 4200+ (2 Core)
  cpu MHz : 1000.000
  cache size : 512 KB
  csews43.cse.iitk.ac.in (4 Core)
  Intel(R) Core(TM) i7 CPU 930 @ 2.80GHz 8192 KB cache

- csecourses2.cse.iitk.ac.in (16 Core Machine)
  model name : Intel(R) Xeon(R) CPU E7330 @ 2.40GHz
  cpu MHz : 2393.894
  cache size : 3072 KB

- csecourses3.cse.iitk.ac.in (24 Core Machine)
  model name : Genuine Intel(R) CPU @ 2.66GHz
  cpu MHz : 2659.980
  cache size : 16384 KB

We compare results on a relative scale as well as show absolute performance improvements by comparison with the Cholesky implementation available in ALGLIB 3.1.0, a Cross-platform numerical analysis and data processing library.

4.2.1 Algorithm 1: Sub-Matrix Update

Given a nave sequential algorithm, our approach was to parallelize it using the constructs available in the Intel TBB library. The particular implementation of the sub-matrix update algorithm required us to use the following constructs:

- Parallel For
- Task
  - task::set_ref_count()
• Affinity Partitioner

Since the iteration space as required in the algorithm is not rectangular (it is triangular), we preferred to spawn multiple tasks using the inbuilt task scheduler. Standard fork-join style was taken. There was no need to use continuation passing style because of the inherent non-recursive nature of the algorithm.

• Relative Speedups:
  Firstly we compare the number of times the naive sequential algorithm could be sped up by using parallelization techniques. The following graph illustrates the results:

![Fig 9: relative speedup for TBB implementation of submatrix update.](image)

Here we see that the single core machine, as expected, is unable to produce any speedups, rather, the amount of overhead associated with the scheduling makes the parallel code even slower than the sequential code. In all other cases we find that the speedup increases and then converges to a constant value. On a dual core-machine, the code could be executed twice faster, on a 4 core machine 4 times. Machines with higher cores (16 & 4) show linear increase in speeds with a possible & unverified convergence to a constant value which lies beyond the graph.

Note that the speedup values do not say that you are able to obtain absolute speedups of order 2 on a dual-core machine. It simply says that a very naive un-optimized code could be made to execute twice faster on machine having 2 cores. Later well see speedups that go beyond the number of cores available in the hardware!

• Absolute Results:
  Having looked at speedups, we shall show the actual execution times of these parallel code.
In this case, we see that the single core-machine out-performs the dual-core machine. However, this doesn't mean that the dual core machine couldn't obtain any speedups. This simply is because the dual core machine is slower than the single core machine. Another case to consider is that the performance of the 4 core machine is the best among the group. This is simply because the 4 core machine is faster than the higher core machines. The behaviour is normal for 16 & 24 core machines.

4.2.2 Algorithm 2: Inner-product update

Again, our approach was to parallelize the naive sequential algorithm. Since at each sub-stage of the algorithm, the iteration space turns out to have a rectangular shape, there was no explicit need to spawn tasks. Everything could be implemented solely by using the in-built parallel for.

- **Relative speedups:**
  First, we consider the speedups associated with this algorithm.

  We find that the speedups obtained by the single core machine is again below 1. Dual and quad core machines provide almost the same speedups, with the 8 core machine showing more variations, possibly because of work-load. The cause of slight dip at problem size of 1600 in all the curves is yet to be identified. In general, significantly higher amount of speedups can be obtained if the number of cores of the machine are pushed a bit further. However, something to note is that the speedups is on a constant rise in case of the 16 & 24 core machine. This can be obtained in 2 ways.
Fig 11: relative speedup for TBB implementation of inner-product subtraction.

1. Slowed execution of sequential un-optimized code as problem size increases.
2. Faster execution of parallel code as problem size increases.

Amongst the 2 possibilities, the former is the case. The sequential code is slowed down significantly because of recurrent cache missed as a consequence of the un-optimized nature of the code. Nevertheless, we see that the curve for 16 core seems to start to converge to a speedup value, but still more improvements can be obtained from that of 24 core machine which hasnt started to converge as of yet.

• Absolute result:
Again, the execution time would provide a better insight into what is actually happening at an absolute scale.

Here the results seem more normal, depicting the expected behaviour of the problem. As the number of cores are increased, the execution time of the problem goes down. There are no upsets. Significant improvements can be seen in all cases.
4.2.3 Algorithm 3: Blocked cholesky

The optimization process for the blocked cholesky case is a bit complicated. Speaking broadly, this can be decomposed into the following steps:

- Given a naive sequential implementation
  - Implement Blocked Cholesky
  - Optimize Sequential Code
    * Loop Interchange (LI) to find optimal loop ordering
    * Common Sub-Expression Elimination (CSE) + Constant Propagation (CP)
    * Loop-Invariant Code Motion (LICM) + Strength Reduction (SR)
  - Optimize block-size.
  - Parallelize optimized sequential blocked implementation.

One issue to be dealt at this stage is to decide on the data packing format. We chose square blocked full row-major order throughout the experiment. For the algorithm we chose the LAPACK POTRF. The algorithm is summarized below:
Algorithm 3 LAPACK POTRF (blocked left-looking algorithm)

1. for $j = 1$ to $n/b$ do
2. partition matrix so that diagonal block $(j, j)$ is $A_{22} = \begin{pmatrix} A_{11} & * & * \\ A_{21} & A_{22} & * \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$
3. update diagonal block $(j, j)$ (SYRK): $A_{22} = A_{22} - A_{21} A_{21}^T$
4. factor diagonal block $(j, j)$ (POTRF): $A_{22} = \text{Chol}(A_{22})$
5. update block column (GEMM): $A_{32} = A_{32} - A_{31} A_{21}^T$
6. triangular solve for block column (TRSM): $A_{32} = A_{32} A_{22}^{-T}$
7. end for

Fig 13: Blocked Algorithm

1. Standard Compiler optimization techniques:
The first step involved using standard compiler optimization techniques to the code. Use used, loop interchange (LI) to find optimal loop ordering. After that common sub-expression elimination (CSE) and constant propagation (CP) techniques were used, followed by loop-invariant code motion (LICM) and strength reduction (SR). The results are as follows:

![Execution-time vs Problem Size](image_url)

Fig 14: Performance improvements through various optimizations techniques.

Across the graph we see constant improvement as we keep on optimizing the code. A notable feature of the graph is the behaviour of blocked code vs unblocked code. Initially blocked code results in
increased execution times, but with an increase in the problem size, we see improvements and the blocking pays off. The most optimized code could perform about 2 times faster on a sequential 1 core machine.

2. Optimize Block Size for hardware:
The next task is to optimize the code for the block size. We obtained the following results.

![Fig 15: Optimal block size for various machines.](image)

This is a typical sawtooth curve. The execution time sharply increases as we increase the block size, and then falls sharply. This is attributed to the padding that goes along with the packing of matrix into square blocked full row-major format. There is sharp rise because of the redundant operations that are performed, that too on a partially filled block.

- Relative Speedups:
Next we compare speedup results. The following graph shows the parallelization benefit of a sequential blocked code, i.e., we are parallelizing a blocked implementation & calculating speedups using the sequential blocked (not the un-optimized code).

Here we find that the behaviour of curves is very similar to that obtained in the unblocked inner-product update case. We conclude that the behaviour is determined on a macro level based on the choice of the algorithm and blocking doesn’t produces much impact to it. The algorithm is still inner-product update after all.
Fig 16: Relative Speedup for different cores.

- **Absolute Results:**
  Comparing the execution times.

Once again one has to consider the slowness of dual-core machine before saying that blocking produces negative impact here. Here, speedups are obtained for dual-core machine, though it still performs worse than the single core machine. Rest of the results are as expected.
4.3 Part 3: Normalized TBB results

We compare the results obtained till now with the sequential implementation in ALGLIB, which implements a cache-oblivious recursive cholesky solution. Henceforth, we assume that it is the best sequential solution.

4.3.1 Dual Core Comparison:

Fig 18: Comparison between algorithms for dual core machine.

On a dual-core machine, we could obtain performance better than ALGLIB. The parallel inner-product algorithm only could perform better than ALGLIB. Others still gave poorer performance in comparison.

Fig 19: Comparison between algorithms for Quad core machine
4.3.2 Quad Core Comparison:

Here, the blocked version performed significantly poorer than the serial ALGLIB. Both other parallel solutions could beat the ALGLIB performance. Overall, the best forming algorithm was parallel-sub-matrix update.

4.3.3 16 Core Comparison:

![Graph showing comparison between algorithms for 16 core machine.](image)

Fig 20: Comparison between algorithms for 16 core machine.

This is where the results are most promising. Serial ALGLIB performs far poorer than the parallel versions of the algorithms (though it performed better for small problem sizes where parallelization couldn’t show significant benefits because of large overhead to computation ratio). An interesting point to note is that blocking an inner-product algorithm hasn’t given any improvements at all! On the contrary, there is some overhead attached to it. In general, parallel inner-product subtraction algorithm was found to be the best performing for this particular hardware.
4.3.4 24 Core comparison:

![Graph showing comparison between algorithms for 24 core machine.](image)

Fig 21: Comparison between algorithms for 24 core machine.

The cause for the sharp rise at problem size 500 is yet to be identified. It required larger problem sizes for the parallel sub-matrix update to perform better than the serial ALGLIB. Otherwise, the behaviour is more or less similar to the 16 core case.

5 Conclusions

We found that going in depth and optimizing a particular hardware can significantly boost the performance and reduce execution times, as evident by the standard compiler optimization techniques used by us, which gave us twice better performance. Further, parallelization can also achieve significant speedups if done properly. The algorithmic choice is an important one to make, because the execution time is hugely determined by the underlying algorithm.

AN interesting conclusion that we arrived at, after interpreting the results was that the introduction of blocked operations did not introduced and significant improvements in terms of execution time over traditional unblocked version. This could possibly be because of smart scheduling & chunking by the Intel TBB runtime.

6 Future Work

1. We observed the impact of changing the block size on execution times by calculating the execution time for all possible block sizes. A better algorithm would have identified & generated only the lows of the saw-tooth pattern to find the optimal block size.
2. The upper half of matrix is not used in blocked implementation. Hence the code can be modified to produce a near-optimal (space optimality) solution without any impact on the performance.

3. We studied performance for both GPGPU & multi-core architecture. A heterogeneous solution can be found by combining both solutions. The basic approach would be to execute partially on GPU, and then finish the remaining on multi-core CPU. The optimal partial execution stage for the GPU could then be found by testing.

References


[2] Parallelizing Choleskys decomposition algorithm - Sylvain HENRY.


