Some Optimization Techniques of the Matrix Multiplication Algorithm

Nenad Anchev, Marjan Gusev, Sasko Ristov, and Blagoj Atanasovski
Ss. Cyril and Methodius University, Skopje, Macedonia
Faculty of Computer Sciences and Engineering
E-mail: nenad_ancev@hotmail.com, {marjan.gushev, sashko.ristov}@finki.ukim.mk, blagoj.atanasovski@gmail.com

Abstract. *Dense matrix-matrix multiplication algorithm is widely used in large scientific applications, and often it is an important factor of the overall performance of the application. Therefore, optimizing this algorithm, both for parallel and serial execution would give an overall performance boost. In this paper we overview the most used dense matrix multiplication optimization techniques applicable for multicore processors. These methods can speedup the multicore parallel execution focusing on reducing the number of memory accesses and improving the algorithm according to hardware architecture and organization.*

Keywords. HPC, CPU, Cache, Performance

1. Introduction

Modern computer systems are far from simple. The basic Von-Neumann's idea has evolved through the decades with the purpose of gaining more performance by adding less hardware. As stated in Moore's law, the evolution of processor and memory systems has brought a possibility to obtain greater computational speeds, but complicating the hardware design.

This complex hardware makes problems to achieve the maximum performance. The existing design bottlenecks make the applications to be far away from maximum performance specified by maximum hardware speed. Therefore we usually have to be satisfied by getting near-optimal performance from the hardware, at an expensive price [1].

The price we have to pay for this performance is building a complex algorithm that resolves the known bottlenecks of the specific architecture. And this algorithm will only show its best results only on the considered processor architecture. A minor architectural change will usually result in sharp performance drop.

Furthermore, modern computer systems use massive parallelism and we have to consider the parallel algorithm optimization. Issues coming from the use of shared resources or communication between the threads have to be carefully resolved, without creating additional bottlenecks or even deadlocks.

However, having a supercomputer, grid or cluster is not enough to achieve the speedup for a particular algorithm. The algorithm should be granular and scalable in order to utilize the hardware resources well. Dense matrix-matrix and matrix-vector multiplications are such algorithms since no data dependency exists and each processor can work with its own data.

Dense matrix-matrix multiplication is a typical scenario in scientific applications. There are many papers which propose improvements of this algorithm or even new algorithms. Here, we analyze the simplest of these algorithms presented in Listing 1 since it can be easy parallelized.

Listing 1: The simplest matrix multiplication algorithm

```
for(i = 0; i < N; i ++)
  for(j = 0; j < N; j ++)
    for(k = 0; k < N; k ++)
      c[i * N + j] += a[i * N + k]
                    * b[k * N + j];
```

The performances of this algorithm are far from optimal while executed on a single-core processor. Regardless of its speed, there is a significant room for improvement. Its bottlenecks will be presented here, one by one. Furthermore, there is a need for an efficient parallelization in order to efficiently exploit the performance of a multi-core, multi-chip machine, or even multi-computer.

The rest of the paper is organized as follows. In Section 2 we explain the algorithm and the main bottlenecks that appear in parallel execution in modern processors. Section 3 proposes well

known techniques to optimize the parallel execution. Section 4 concludes our work and presents our plans for future work.

2. Algorithm Bottlenecks

The bottlenecks of the sequential algorithm and the parallelization challenges are presented in this section.

2.1. Redundant Operations

Reducing the number of executable machine operations in the algorithm execution is a great challenge. In this case we aim not to change the arithmetic operations leading to change of the algorithm itself or its structure. We are targeting at the unnecessary multiplications to compute the element address i.e. the "$(i * N)$", which is executed in a loop, even though its value is unchanged throughout the iterations.

Especially this refers to reducing the number of memory accesses. A typical example is the implicit fetching of the matrix C. As the result matrix is not initialized, we will not have to read its value. Moreover, we would like to calculate the final value of each element and use only one write per each element.

2.2. Short Loops

Modern processors tend to use deep pipelines, often 20 or more levels of depth [2]. This could be an obstacle in programs which have high frequency of branch instructions, as in our case. If we translate the "single" one line instruction in the nested loops, it will have 11 simple arithmetic instructions and 3 memory accesses, together with the unnecessary ones. This will cause the processor pipeline to be underutilized by emptying it even before it is actually full, degrading the algorithm performance [3].

2.3. Cache Usage

The worst bottleneck in this algorithm is the irrational cache usage. When matrices become larger, cache memory becomes too small to keep them whole inside, and cache misses are generated. This problem is known as cache capacity problem [4]. However, cache capacity is not the only problem that occurs within caches. Cache replacement policy also impacts the algorithm performance [5].

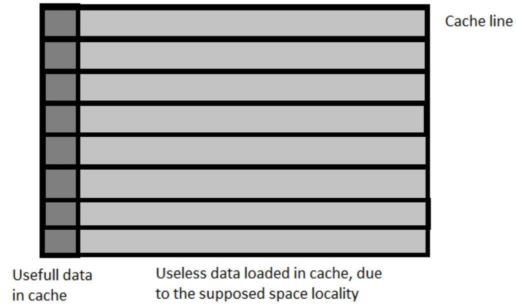

Figure 1: Non transposed second matrix

As dense matrix-matrix multiplication algorithm is memory-intensive and more importantly - cache intensive [6], we should try to optimize it by using the cache hierarchy as effectively and efficiently as possible. Cache associativity is also a huge problem, since only a small number of sets may be used for particular column of the second matrix [7].

The cache misses in this algorithm comes from many reasons. The first was described earlier, i.e. unnecessary implicit reading and storing the result matrix elements in the cache.

The second point where we could gain performance is the strided memory access of the second matrix, illustrated in Figure 1. Strided memory access means that the space locality supposed by the cache memory is not used in the expected way. As matrices in program language C are represented in row-major order, a serious cache memory leak occurs. When we fetch a cache line from the main memory, we use only one element of it, thus filling the cache with unnecessary data.

The third point producing performance drawback occurs when the cache is not enough to hold even one matrix [4]. In this case, the same parts of the matrix are loaded in the cache multiple times, thus producing unnecessary expensive communication with the main memory. Reducing the memory access by improving the usage of the data that is already stored in the cache should improve the performance.

2.4. Parallelization challenges

The simplest and most efficient method to parallelize this algorithm is to distribute the computations equally to all available processors. This would be a fairly easy task, but we should consider the cache architecture of the processor cores when we distribute the computation on each of the cores.

The task distribution should also care about efficient cache usage, having in mind the dedicated and shared levels of the cache hierarchy. Dedicated cache per core means more cache available at the system, but shared cache has an advantage of providing the result of one expensive load for all cores sharing the cache [8]. It also provides superlinear speedup region for parallel implementation [9].

3. Optimization Techniques

Knowing the bottlenecks of the algorithm, we will propose methods and techniques to resolve them and boost the performance. We will also explain our view for an efficient parallelization of this algorithm.

3.1. Operation Reduction

Reducing the redundant operations is the easiest task described here. The only thing that needs to be done here is to push the multiplications in the outermost loop possible. This should directly reduce the processor cycles that were wasted in repeating the same computation . The other thing that can be done here is to modify the calculation of the final result. The intermediate sums can be stored in a temporary variable, which should be written only once to the result matrix, when it has the final value. This will also reduce the need to keep part of the result matrix in the cache and the number of memory writes.

3.2. Loop Unrolling

This technique is used to solve two of the presented issues: the problem of short loops and to a lesser extent, it will reduce the number of operations. We propose only a partial loop unrolling, with just a few iterations of the innermost loop unrolled. This will produce enough straight-line assembly code for a more efficient pipeline usage. Furthermore, it will reduce the number of operations by reducing the branch instructions by times of the unroll factor. Partially unrolled loops can also be tuned to process as much elements as a cache line contains, and thus be combined with the prefetch technique. However, this point would be further discussed in the next Section 3.3.

3.3. Transposition

Transposition has the task to deal with the strided memory access of the second matrix. As explained in the previous section, the cache space locality is not used for the caching of the second matrix, and the cache is filled with useless data.

If we transpose the second multiplier matrix, we could fix the problem of strided data access. Transposing the matrix before the multiplication, adds additional operations and memory access, but increases the performance. This "anomaly" could be explained as following: Matrix multiplication memory complexity is $O(N^3)$ for each matrix. Because of the strided access, the second matrix produces cache misses more often, which generally makes N^3 "slow" memory accesses. This is much more visible for larger matrices.

When we have the second matrix transposed, the N^3 "slow" operations are speeded up, because cache is now used more optimally, as it is not filled with unnecessary data. This produces visible speedup. On the other hand, the transposition creates additional "slow" operations. Transposing a matrix, means that either the reading, or the writing should be strided. Results show that strided write produces better performance than strided read. [1]. But the number of operations needed to transpose a matrix is N^2, which means that we add N^2 slow operations to speedup N^3 operations, which seems rational for larger matrices. Jenks [10] found superlinear speedup (speedup greater than the number of processors) with parallel execution of matrix multiplication algorithm using MPI and transposing one matrix.

3.4. Blocking

Blocking is a well-proven technique which boosts the performance of the matrix multiplica-

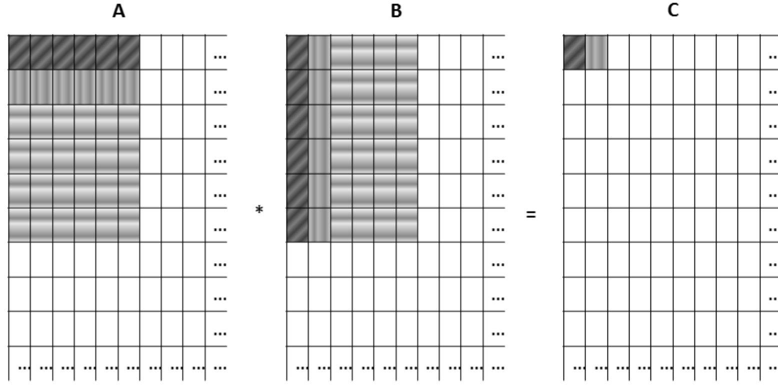

Figure 2: Blocking matrix multiplication [11]

tion algorithm. It deals with optimizing the cache usage by reducing the memory access. In few words, blocking means that we turn a memory-intensive algorithm into cache-intensive, by using the data already loaded in the cache as long as can be used. The algorithm multiplies submatrices or blocks of matrices with same size b instead of multiplying the entire row of the first matrix with the entire column of the second. Figure 2 depicts the algorithm. It means that after the data is removed from the cache, all the computations on it are already done from the cache, and it won't need to be loaded again. This would be a result of using an optimal blocking algorithm. Another thing that we should care when using blocking is to fill the cache with data size equal to the cache size. Larger data sets will mean that certain cache line would be replaced, which we try to avoid, and smaller data sets won't use the maximum available performance [11].

In parallelized algorithm, blocking should also care about the cache sharing between cores. It should keep the shared data in the shared cache, so that all of the cores that share this data will use it. It should also care that cores won't compete about shared cache with their private data, which would mean mutual overwriting of private data by the cores.

3.5. Prefetch

This technique is used to hide the memory latency. Modern processors have multiple execution units, which means that data access and computation will be realized by separate subunits in the processor. [2] This gives us the idea to prefetch the data necessary for the next instructions, while the current instructions are executing. It will reduce the memory latency, and the waiting cycles of the execution units.

As we proposed to transpose the second matrix before multiplying it, we will analyze the prefetch used both in the transposition, and in the multiplication algorithm.

Combining blocking and prefetching can improve the performance of transposition algorithm up to 5 times at PowerPC architecture [12]. As we don't have data reuse (every element is read and written only once), blocking wouldn't help much here, and this improvement can be greatly attributed to prefetch. By further applying the improvements to the drawbacks due to cache associativity and cache line [7], there is a room for further improvement here. Williams et al. [13] used padding to the first element of each submatrix to land on equidistant cache sets.

Prefetching could also improve performance at the multiplication part itself. However, we should not expect such a great improvement here, because the data from the cache is reused (by using ideal blocking, every element in the cache would be used N times). This means that we already have optimized the access time for the (n-1) times the element is used. The expected memory access speedup in the multiplication part, if a perfect blocking algorithm is used is defined in (1), where EMS denotes for *Expected Memory Speedup*, MAT denotes for *Memory Access Time*, CAT denotes for *Cache Acces Time* and $ATERC$ denotes for *Average Times Of Element Reloading In Cache*

$$EMS = \frac{1}{N} \cdot \frac{MAT}{CAT} \cdot ATERC \qquad (1)$$

Prefetching could only improve the first time the element is read from the main memory, which means that we should not expect larger improvement here. Any improvement should be expected for very small matrices, where $(1/N)$ would be larger, and for very large matrices where only small parts of them can fit into the cache. Here, even by using blocking, there would still be a need of frequent memory access, and prefetching would lead to some improvement here.

A sort of implicit prefetch occurs when we have a shared cache. When one core accesses data needed for all of the cores, it implicitly makes a prefetch for the other cores, which won't have to make an expensive load from the main memory [8].

3.6. Linear algebra (BLAS, LAPACK, ATLAS, ...)

There are special highly optimized implementations of matrix-matrix multiplication algorithm which produce maximum performance for their target architecture. However, these algorithms usually differ from the basic algorithm analyzed here. Some authors [14] propose their own implementations of these algorithms, which will further boost the performance. But these implementations differ from the concept of simple matrix-matrix multiplication, and the optimization techniques used there would be different from those analyzed here. The analysis of some of these algorithms will be part of our future work.

3.7. Data-flow computing

One of the promising, but still not widely used methods, is the use of data-flow computing engines as a matrix multipliers. This approach would give us the freedom to construct a matrix multiplier as a complete hardware solution Knowing the algorithm's requirements, our ideal data-flow hardware would avoid most of its bottlenecks described here.

The whole programming concept is different here. There would be no redundant operations, except if we don't create them accidentally. Short loops wouldn't create any drawbacks, and we would have an explicit control over the memory and the caching. The level of parallelism would be the number of parallel multipliers and adders we create, and the way we interconnect them. Overall, we would have direct control over the constructed hardware, which promises a slightly improved performance.

However, the price we have to pay for this will be paid in programming effort. The concept of data-flow programming is still in an initial phase, and requires a paradigm shift, so the potential problems and drawbacks that may appear for the ordinary programmer are still unknown.

4. Conclusion and Future Work

We have presented the main bottlenecks of the simple dense matrix-matrix multiplication algorithm. We have also made an overview of the common optimization techniques for matrix multiplication algorithm, and the things we have to care when parallelizing this algorithm.

All techniques for optimization that speed up the algorithm execution can be classified in three main classes:

- Optimization of the algorithm by changing the order and the number of operations - Partial loop unrolling, reducing operations, etc;
- Optimization according to memory resources - Transposing the second matrix, Prefetching, Optimizing cache usage, Blocking; and
- Using parallel implementation - using the features of granularity and scalability of the algorithm.

Most of the techniques here are presented in many other papers, which usually concentrate on a single technique and the performance obtained by it. Here, we made an overview of them, and showed the advantages and the challenges of combining them together. Our intention here is to make a guideline of practical optimization and parallelization of this algorithm, and which characteristics of the target architecture should be considered when implementing

this algorithm. The points presented here may be a basic guideline of optimization of similar memory-intensive algorithms. This analysis can be used for other similar algorithms, of linear algebra, such as matrix-vector multiplication.

Our future work will consist of implementation of presented techniques for optimization to basic matrix multiplication algorithm. We will analyze the performance boost provided by each of these techniques, and the overall speedup gained when they are used in combination. The proposed analysis would be realized by tuning the algorithm for different processors with different cache architecture and organization (different cache size, cache line, cache levels, cache associativity, cache replacement policy, etc), for the purpose of analyzing which cache level is crucial for the performance. A dataflow computing implementation will also be part of our future research. We also plan to analyze the BLAS implementations of this algorithm, and the techniques used at those implementations, for the purpose of obtaining even greater performances.

References

[1] G. Hager and G. Wellein, "Introduction to high performance computing for scientists and engineers," USA, 2010.

[2] V. Milutinovic, "Issues in microprocessor and multiprocessor systems," 1999.

[3] G. Hager, T. Zeiser, J. Treibig, and G. Wellein, "Optimizing performance on modern HPC systems: Learning from simple kernel benchmarks," in *Computational Science and High Perf. Computing II*. Springer Berlin Heidelberg, 2006, vol. 91, pp. 273–287.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*. USA: Elsevier, 2012.

[5] N. Anchev, M. Gusev, S. Ristov, and B. Atanasovski, "Optimal cache replacement policy for matrix multiplication," in *ICT Innovations 2012*, S. Markovski and M. Gusev, Eds. Springer Verlag / Berlin Heidelberg, 2013, vol. AISC 257, pp. 71–80.

[6] S. Ristov, M. Gusev, M. Kostoska, and K. Kiroski, "Virtualized environments in cloud can have superlinear speedup," in *Proceedings of the Fifth Balkan Conference in Informatics*, ser. BCI '12. New York, NY, USA: ACM, 2012, pp. 8–13.

[7] M. Gusev and S. Ristov, "Performance gains and drawbacks using set associative cache," *Journal of Next Generation Information Technology (JNIT)*, vol. 3, no. 3, pp. 87–98, 31 Aug 2012.

[8] M. Jacquelin, L. Marchal, and Y. Robert, "Complexity analysis and performance evaluation of matrix product on multicore architectures," in *Proceedings of the 2009 International Conference on Parallel Processing*, ser. ICPP '09. USA: IEEE Computer Society, 2009, pp. 196–203.

[9] S. Ristov and M. Gusev, "Superlinear speedup for matrix multiplication," in *Proc. of the 34th Int. Conf. on Information Technology Interfaces, ITI 2012, IEEE Conference Publications*, 2012, pp. 499–504.

[10] S. Jenks, "Multithreading and thread migration using MPI and myrinet," in *Proceedings of the Parallel and Distributed Computing and Systems*, ser. PDCS'04, 2004.

[11] M. Gusev, S. Ristov, and G. Velkoski, "Hybrid 2d/1d blocking as optimal matrix-matrix multiplication," in *ICT Innovations 2012*, S. Markovski and M. Gusev, Eds. Springer Verlag / Berlin Heidelberg, 2013, vol. AISC 257, pp. 13–22.

[12] G. Mateescu, G. H. Bauer, and R. A. Fiedler, "Optimizing matrix transposes using a power7 cache model and explicit prefetching," *SIGMETRICS Perf. Eval. Rev.*, vol. 40, no. 2, pp. 68–73, Oct. 2012.

[13] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Par. Comput.*, vol. 35, no. 3, pp. 178–194, Mar. 2009.

[14] R. A. V. D. Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," Tech. Rep., 1997.