Intel vs AMD: Matrix Multiplication Performance

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

***Abstract*—Matrix-Matrix multiplication (MMM) is widely used algorithm in today's computations and researches. Many techniques exist to speed up its execution. In this paper, we analyze the performance of MMM varying matrix size in order to determine its behavior and the region where it provides the best performance. We also determine the best speedup and efficiency in parallel implementation for different CPU architectures since cache architecture and organization is very important for MMM performance. Intel i7 and AMD Opteron CPUs are used as an environment. Several achieved results are expected, but there are also many unexpected. Superlinear speedup (speedup greater than the number of used threads) and the efficiency greater than 100% are achieved for each parallel implementation only on AMD Opteron. We observe regions with performance discrepancy for all three parameters for both CPUs.**

***Index Terms*—HPC, CPU, Cache, Memory, Superlinear Speedup**

I. Introduction

MMM is a linear algebra algorithm which has been a subject of research in industry and the science for a long time. There are many techniques to speedup the algorithm: using faster processors, modifying the algorithm for less operations, using different hardware architecture. Since it is quite well for parallelization (granular and scalable), huge speedup can be achieved if it is scaled on many cores.

Various CPU vendors produce different CPU architectures. In this paper we realize a systematic approach to analyze the performance of two different CPUs, i.e. Intel i7 and AMD Opteron using the same MMM algorithm. We choose these CPUs since both have similar cache architecture. Both use four cores with private L1 and L2 caches, and all cores share the last level L3 cache.

Our goal is not to improve the "slow(est)" dense MMM, but we use it intentionally since it generates a lot of cache misses and is a good benchmark for many cache parameters: cache size, cache levels, replacement policy, set associativity, cache line size, cache inclusivity / exclusivity etc. We measure the response time for each test case and calculate the speed, and speedup and efficiency in parallel executions.

The rest of the paper is organized as follows: Section II describes the related work. In Section III we describe the testing methodology used in our experiments. The next Section IV elaborates the results using three parameters: speed, speedup and efficiency for both platforms. We discuss the results in Section V. Finally, Section VI concludes our work and Section VI presents our plan for further work.

II. Related Work

MMM algorithm is the most common used algorithm in computations, as well as in research. Many authors analyze their performance trying to speedup its execution with different techniques classified in two main categories improving the algorithm by reducing the operations or improving the algorithm according to available multiprocessor, especially its cache architecture and organization, since matrix multiplication is cache intensive algorithm [1], i.e. each element is reused N times, where N is the the matrix size. Both techniques are important. In this paper we try to understand the behavior of the MMM algorithm presented with the three most important parameters: speed, speedup and efficiency, while the MMM algorithm is executed on two different single-chip multi-core multiprocessors with the same cache structure.

A. Speed Analisys

We found many papers that improve the algorithme according to the available hardware. Hennessy and Patterson [2] present 2D blocking matrices techniques as one of several cache optimization that enormously decreases cache misses and thus the execution time. Although this techniques increase the operations, it reduces the high cache level misses and accesses to main memory since block sizes are chosen as the matrices can be stored in L1 cache. Gusev et al. even improve the 2D blocking matrices by using rectangles instead of squares. They reduce the performance drawbacks due to associativity [3] for the processors with small associativity (AMD) by reducing the height of the second matrix B blocks. Williams et al. [4] used padding to the first element of each submatrix to land on equidistant cache sets and thus reduce the drawbacks due to cache set associativity.

B. Speedup Analisys

Achieving linear speedup when the algorithm is scaled is imperative. However, even superlinear speedup is achieved by many authors, but without detailed explanation [5], [6]. Al-Jaroodi et al. [7] found superlinear speedup for matrix multiplication explaining that more processors have more cache and thus parallel execution will generate smaller number of cache misses. However, they do not explain why superlinear speedup is achieved just for 2 and 3 processors, and not for 4 and more processors where more cache capacity is available. Kolberg et al. [8] found superlinear speedup for MMM on 16 processors using MPI, but not for 32 or 64, despite increased cache memory. More detailed but also incomplete analysis is performed in [9] and [10].

Ristov and Gusev [11] introduced more detailed analysis about superlinear speedup. They proved experimentally that superlinear speedup region exist and scaling the MMM on more cores choosing the matrix size from the superlinear region will lead to superlinear speedup regardless of the number of processors. Superlinear speedup is achieved in cloud virtual environment and even on Windows platform in Windows Azure [12], despite the virtualization layer. It can be achieved on multi-GPU implementation [13] due to configurable cache memory of Fermi architecture, as well. Jenks in [14] found superlinear speedup with parallel execution of matrix multiplication algorithm using MPI and transposing one source matrix, thus reducing the cache misses. Adding parallel overheads in order to Increasing both cache reuse and fine-grained parallelism by adding parallel overheads can lead to superlinear speedup [15].

We determine superlinear speedup for MMM as well. But we found another reason, i.e. implicit prefetching due to shared last level L3 cache.

C. *Efficiency Analisys*

We have not found any paper that analyzes the efficiency while scaling the MMM. In this paper we will try to understand if scaling the resources will increase or decrease the efficiency on both multiprocessors. We have set a hypothesis that the efficiency decreases when the resources are increased for the same problem size.

III. Testing Methodology

This section presents the testing methodology used for the experiments in order to provide reliable results.

A. *Testing Algorithm*

We use Dense MMM algorithm with squared matrices $C_{N \cdot N} = A_{N \cdot N} \cdot B_{N \cdot N}$ as test data. Matrices elements are stored as double precision numbers with size $ME = 8$ bytes each. Each element c_{ij} of matrix C is calculated as inner product of row i of matrix A and column j of matrix B, for each $i, j = 0, 1, \cdots, N-1$. Cache miss occurs if an element is not present in the cache and the processor needs to load it from main memory, which is much expensive operation. Increasing the matrix size N will occupy the cache faster, thus increase the cache miss ratio and the total execution time.

We are not interested in speeding up the algorithm by reducing the number of operations, but we use this algorithm since it is cache intensive.

One thread in sequential implementation multiplies the whole matrix $A_{N \cdot N}$ and matrix $B_{N \cdot N}$. For parallel implementation, each thread multiplies the row block matrix $A_{N \cdot N/P}$ and the whole matrix $B_{N \cdot N}$, where $P \in \{2, 3, 4\}$ denotes the total number of parallel threads and used CPU cores. Both implementations are executed without any optimization or adaptation to a certain CPU architecture.

A single thread in sequential implementation executes N^3 sums and N^3 products, or total $2 \cdot N^3$ operations. Each thread in parallel implementation executes average $2 \cdot N/P \cdot N^2$ operations. This means that the algorithm performs $2 \cdot N^3$ operations, both for sequential and parallel execution.

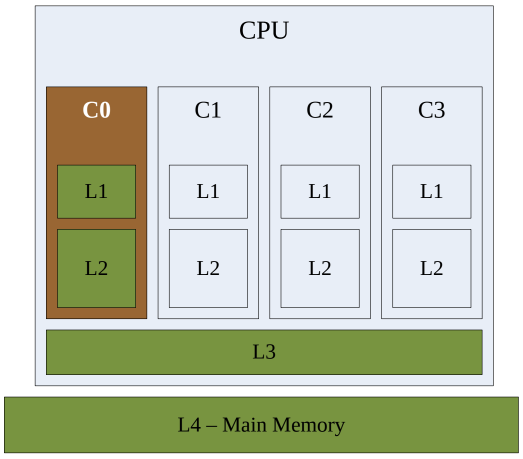

Fig. 1. CPUs cache architecture

CPU	L1D	L1I	L2	L3
Intel i7	32KB	32KB	256KB	8MB
AMD Opteron	64KB	64KB	512KB	2MB

TABLE I
Cache size for i7 and Opteron CPUs

B. *Testing Environment*

Our testing environment consists of two different vendor CPUs, i.e. Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz and Quad-Core AMD Opteron(tm) Processor 8347, each with 4 cores, as depicted in Fig. 1. Each core has its own private L1 and L2 caches and all 4 cores share L3 cache. Table I presents the details about cache size of both CPUs, where L1D and L1I denote for L1 data and instruction cache, correspondingly.

Both servers are configured with the same runtime environment: Linux Ubuntu 10.10 operating system is installed on both machines and C++ with OpenMP is compiled with gcc compiler.

C. *The Experiments*

We realize 4 experiments on both platforms. Each experiment consists of series of test cases varying matrix size from $N = 1$ to $N = 1100$.

1) Sequential Implementation - Experiment 1: Sequential execution is realized with one thread on one core C0 as depicted on Fig. 1. This experiment uses C0 core with its private L1 and L2 caches, as well as the whole L3 Cache.

2) Experiments 2, 3 and 4 - Parallel Implementations: Three experiments are realized using 2, 3 and 4 processors. Each processor has its own private L1 and L2 caches, but now share L3 cache.

D. Test Data

We measure the execution time $T(P)$ for each test case (different matrix size N) in each experiment (different number of processors P). We use $T(P)$ to calculate the *Speed* $V(P)$, the *Speedup* $S(P)$ and the *Efficiency* $E(P)$ using well known definitions (1), (2) and (3).

We calculate the speed $V(P)$ by dividing the number of operations and measured execution time as defined in (1), both for sequential and parallel execution, i.e. for $P \in \{1, 2, 3, 4\}$.

$$V(P) = \frac{2 \cdot N^3}{T(P)} \tag{1}$$

The speedup $S(P)$ for parallel execution (for $P \in \{2, 3, 4\}$) is calculated as ratio of sequential and parallel execution time by (2).

$$S(P) = \frac{T(1)}{T(P)} \tag{2}$$

Finnaly, the efficienacy $E(P)$ is calculated for parallel execution (for $P \in \{2, 3, 4\}$) as ratio of the speedup and the number of processors (3). We will use the efficiency to determine which parallel implementation provides the best speedup per core.

$$E(P) = \frac{S(P)}{P} \tag{3}$$

E. Test Goals

The test experiments have two goals. The first goal is to model multiprocessor's behavior, i.e. speed, speedup and efficiency as a function of matrix size for cache intensive MMM algorithm on both environments. The second goal is to determine which platform provides better performance.

F. Testing Regions

Gusev and Ristov [1] define different cache regions L_1, L_2, L_3 and L_4 where matrix multiplication behaves differently. The L_1 region is determined with those matrix sizes that can be stored in L1 cache. Then we expect the highest processor speed and the smallest number of cache misses. L_2 region is determined when the matrices size is such that the matrices can not be stored in L1 cache, but fit in the L2 cache generating cache misses in L1, but not for L2. Respectively in L_3 region L2 generates cache misses but data fits in L3. The region after this point is called L_4 region presenting main memory or Level 4 cache.

IV. Results of the Experiments

In this section we present the results of the experiments realized in two CPUs with the same cache distribution, but different cache size, as explained in Section III-B.

A. Results on Intel i7 CPU

We measure the execution time for each number of processors $P \in \{1, 2, 3, 4\}$. In this section we analyze the results of calculated Speed $V(P)$, Speedup $S(P)$ and Efficiency ($E(P)$ for each experiment defined in Section III-C while executed on Intel i7 processor.

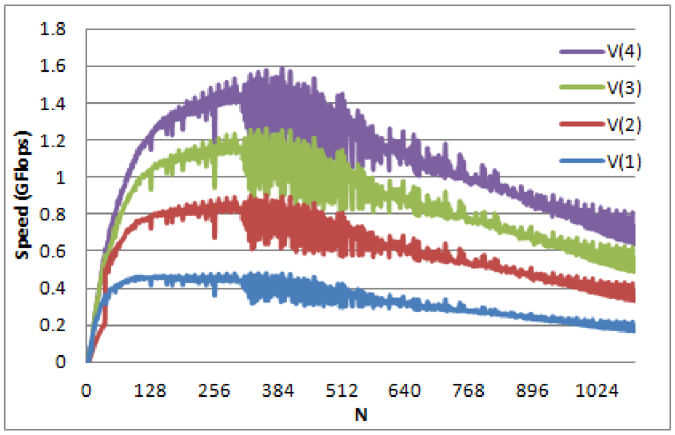

Fig. 2. Speed for MMM as a function of matrix size while executed on different number of Intel i7 cores

1) Speed on Intel i7 CPU: The speed for each test case of all four experiments while executed on Intel i7 processor is depicted in Fig. 2.

We observe three regions (A, B, and C going from left to the right) with different speed behavior for each experiment. The speed increases in region A while N increases until region B where it saturates. Then it decreases in region C which is more emphasized for the experiments with greater number of processors. However, the three regions are different for each experiment and their ranges are presented in Table II.

TABLE II
DIFFERENT SPEED BEHAVIOR

Exp.	Region A	Region B	Region C
1	[1,102]	[102,380]	[380,1100]
2	[1,135]	[135,380]	[380,1100]
3	[1,275]	[275,380]	[380,1100]
4	[1,295]	[295,380]	[380,1100]

Speed drawbacks are observed for $N = 128$ and more emphasized for $N = 256$ for all experiments due to cache set associativity [3]. The drawback is more emphasized while using more processors since all processors force for the same set and more cache misses are generated.

Another unexpected speed behavior is in the region $N = [312, 580]$ where its value is discrepant. Another discrepant region is for matrix size $N > 932$. We believe that this discrepancy happens due to cache line, L2 and L3 cache capacity and associativity. We will analyze these unexpected results in more details in our further research analyzing the cache misses, the address of the first element of matrices and matrix pattern in the cache and main memory.

We can conclude that the maximum performance (speed) can be achieved when the matrices A and B can be fit in L3 cache, i.e. around $N = 380$. After this point, the speed starts to decrease for each experiment.

2) Speedup on Intel i7 CPU: The achieved speedup for the three experiments with parallel implementation compared to sequential execution is depicted in Fig. 3.

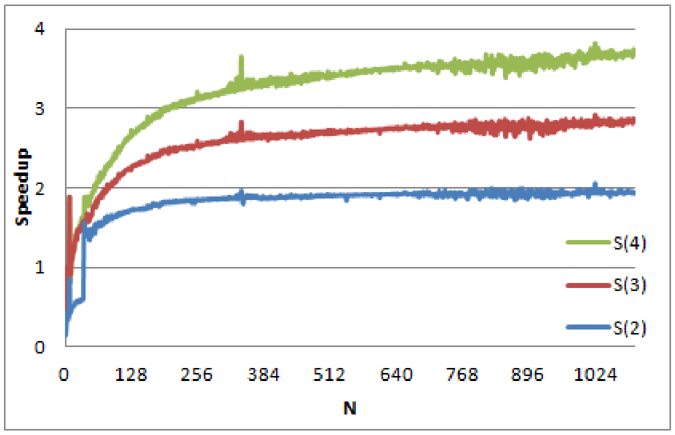

Fig. 3. Speedup for MMM executed as a function of matrix size while executed on different number of Intel i7 cores

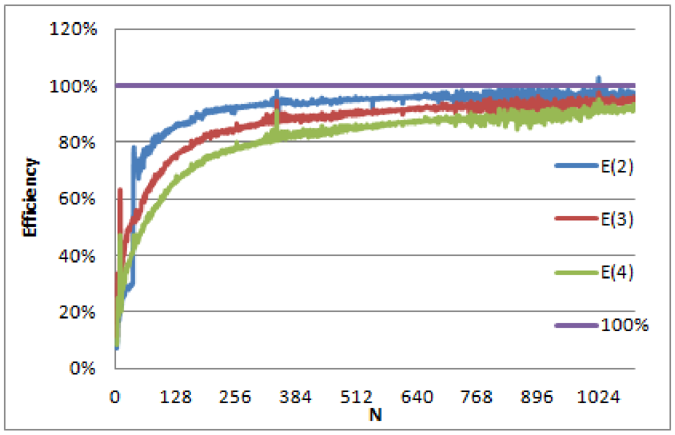

Fig. 4. Efficiency for MMM executed as a function of matrix size while executed on different number of Intel i7 cores

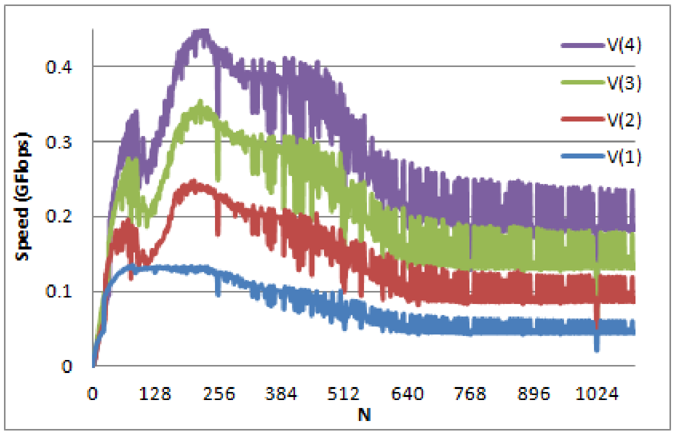

Fig. 5. Speed for MMM as a function of matrix size while executed on different number of AMD Opteron cores

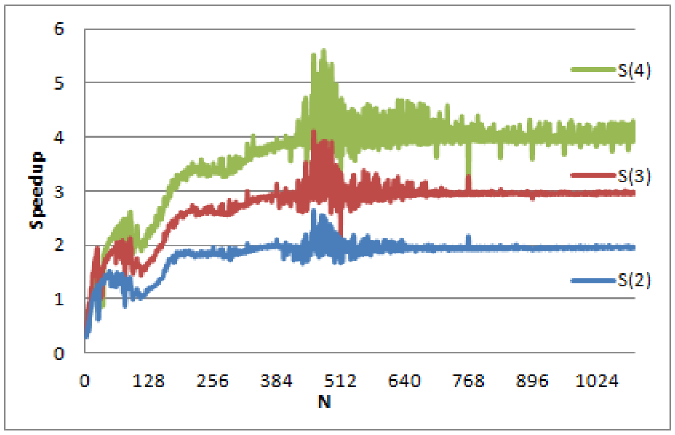

Fig. 6. Speedup for MMM executed as a function of matrix size while executed on different number of AMD Opteron cores

The speedup in the three experiments with parallel implementation satisfies Gustafson's Law [16], i.e. it is sublinear in each test case. The speedup rises and saturates until its limit, i.e. $S(P) \to P$ for each experiment.

Two unexpected results, i.e. a small positive peak for $N = 342$ and the discrepant speedup in the region for matrix size $N > 736$, will be analyzed in more details in our future work.

3) Efficiency on Intel i7 CPU: Fig. 3 depicts the efficiency for the three experiments with parallel implementation. All three experiments have similar curves for the efficiency, where experiment 2 is the most efficient implementation in front of experiment 3 and 4, i.e. $E(2) > E(3) > E(4)$ for each test case. We can conclude that adding more resources reduces the efficiency, and the efficiency increases for greater matrix size N. However, the best choice should be the maximum speed.

B. *Results on AMD Opteron CPU*

In this section we analyze the results of calculated Speed $V(P)$, Speedup $S(P)$ and Efficiency $(E(P)$ for each test case for all experiments executed on AMD Opteron processor.

1) Speed on AMD Opteron CPU: The speed for each test case of all four experiments while executed on AMD processor is depicted in Fig. 5. We observe that the speed for sequential execution behaves different than those for parallel execution.

We can model the $V(1)$ behavior in four regions, i.e. increasing, maximum, decreasing and saturating, while increasing the matrices size N. But very unexpected behavior appears for speed for parallel execution. There are two ranges $N = [81, 84]$ and $N = [217, 218]$ with local maximums. After the second maximum, the speed starts to decrease and then saturates.

Starting from $N = 325$, the speed is also discrepant for AMD CPU, and even more than Intel i7 CPU.

2) Speedup on AMD Opteron CPU: Fig. 6 depicts the achieved speedup for the three experiments with parallel implementation.

We determine a phenomenon for the speedup. Apart of the Gustafson's Law that the maximum speedup is linear, we observe a superlinear speedup regions for each experiment with parallel implementation. Table III presents the regions and maximum values of the achieved speedup. All superlinear regions begin for the same matrix size $N = 435$, but they are greater when more cores are used. Even more, the super-

TABLE III
SPEEDUP REGIONS AND MAXIMUM VALUES

Parameter	Exp. 2	Exp. 3	Exp. 4
Superlinear region	[435, 517]	[435, 657]	[435, 1100]
Max. speedup S_{max}	2.64	4.07	5.58
N_{max} (S_{max} point)	456	456	476

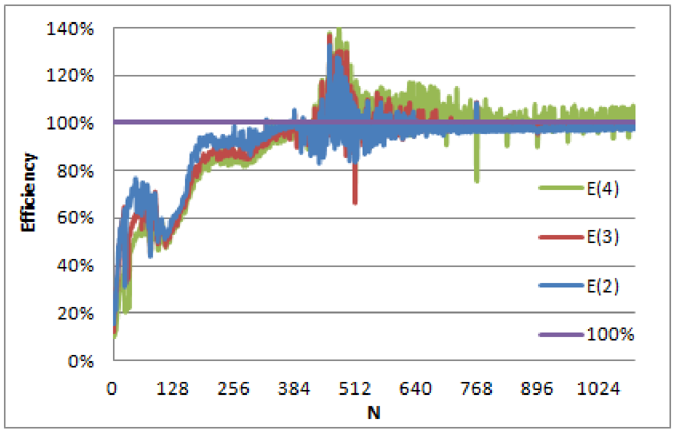

Fig. 7. Efficiency for MMM executed as a function of matrix size while executed on different number of AMD Opteron cores

linear speedup region for 4 cores is infinite (until measured $N = 1100$).

The same local maximums exist as for the speed since $V(1)$ is constant in those regions. And another local speedup maximum appears in superlinear region as presented in Table III.

We can conclude that maximum speedup can be achieved in the region around N_{max} for particular number of cores.

3) Efficiency on AMD Opteron CPU: The efficiency for the three experiments with parallel implementation is depicted in Fig. 7. Two interesting results are achieved. The first, we achieved a regions where the efficiency is greater than 1 (100%) because of achieved superlinear speedup. The second, there are two regions where the efficiency behaves differently. That is, the experiment 2 is the most efficient implementation in front of experiment 3 and 4, i.e. $E(2) > E(3) > E(4)$ for test cases in the region $N < 416$. For $N \geq 416$, the maximum efficiency is achieved for the experiment 4, in front of experiment 3 and 2, i.e. $E(4) > E(3) > E(2)$.

We can conclude that adding more resources reduces the efficiency, and the efficiency increases for greater matrix size N. However, the best choice should be the maximum speed.

V. DISCUSSION

The results show superlinear speedup region for MMM without any optimization on AMD Opteron processor using $P \in \{2, 3, 4\}$ cores.

Let's explain in more details the reasons that lead to superlinear speedup, and only on AMD Opteron CPU. Since both CPUs are on different frequency, we will use CPU clocks. The total number of clocks TC can be expressed as a sum of the clocks that CPU spends on arithmetic operations (CC) and the clocks for memory access (MC), i.e.

$$TC = CC + MC$$

CC is constant for both CPUs while using the same number of cores and therefore it satisfies the Gustafson's Law. The sum of all clocks spent by all cores for parallel execution will be greater than the sequential due to extra operations required for creating the threads and CPU idle state because not all the cores start and finish the execution at the same time.

Let's analyze the MC now. Each matrix element should be accessed N times. The best performance can be achieved if each element is loaded from the main memory only the first time and other $N - 1$ times from L1 cache. The hypothesis set by many authors about superlinear speedup for matrix multiplication is that the reason for superlinear speedup is due to greater capacity of private cache memory per core in parallel execution will generate less cache misses than the sequential execution. However, our results show different reason for superlinear speedup.

We claim that more cache capacity in parallel execution is only one of possible conditions for superlinear speedup achievement. In this paper, we present that the superlinear speedup appears when the problem size can not be stored completely in L3 cache, that is, it does not appear due to greater cache capacity in parallel execution, but the main factor for its existence is the shared L3 cache. This allows a kind of implicit prefetch of the matrix elements. This explains the larger speedup in the main memory region in the shared cache scenario, especially for the experiment with 4 cores. In parallel execution, when one of the cores loads a chunk of a matrix in the cache, it makes an implicit shared cache prefetch to other processors. This scenario is more important in a LRU cache replacement policy, where the matrix B takes more recent hits than the first one A. Therefore, the additional speedup than expected is achieved since the whole matrix B is shared among the cores.

Although we determined another possibility for superlinear speedup existence, it has been achieved only on AMD Opteron CPU, and not on Intel i7 processor, despite the same cache structure (the same number of cores, private L1 and L2 caches and shared L3 cache).

But, let's analyze in more details the speed for Intel i7. According to Gusev and Ristov [1] and the L3 cache size of i7 presented in Table **??**, L_3 region should be until $N = 724$ (two matrices can be stored in L3 cache size of 8MB). However, Fig. 2 depicts that the L_3 region ends around $N = 380$, which seems like L3 cache is not shared among all cores, but only a quarter of 8MB, i.e. 2MB is private per core. This could be the reason why superlinear speed is not achieved on Intel i7.

Despite the nonexistence of superlinear speedup on Intel i7, the speed ratio is much greater than the frequency of the AMD Opteron CPU.

VI. CONCLUSION

This paper analyzes the MMM algorithm executed on two different vendor CPUs, i.e. Intel i7 and AMD Opteron, each with 4 cores with the same cache structure (private L1 and L2 caches per core, and shared L3 cache), but different cache capacities and set associativity. The three most important parameters for MMM are measured and analyzed: the speed, speedup and efficiency varying the matrix size in order determine the different behaviors of the algorithm for different cache regions: L_1, L_2, L_3 and L_4.

The speed as a function of matrix size has similar curves on Intel i7 CPU. All of them have three regions with different speed behavior: increasing, saturating and decreasing. Two regions are detected where the performance is discrepant. However, the achieved speed on AMD Opteron behaves different. Sequential execution is characterized with four regions: increasing, saturating, decreasing and again saturating. The speed for parallel implementations has two local maximums in the first saturating region for sequential execution. The speed is discrepant in the whole region for matrix size greater than 325 for AMD Opteron, and it is even greater than the discrepancy for Intel i7 CPU. The maximum speed for AMD Opteron can be achieved at the end of L_2 region and for Intel i7 in L_3 region.

The speedup as a function of matrix size for each parallel execution complies with the Gustafson's Law for Intel i7. It is greater for greater matrix size and saturates towards linear speedup.

Superlinear speedup is achieved on AMD Opteron for all parallel executions. The main reason is not only due to greater cache capacity to store the matrix elements in parallel execution, because it appears in shared L_3 region instead of private L_2 region. We determine that superlinear speedup appear because the shared last level cache allows implicit prefetching of the matrix elements in parallel execution, i.e. when one of the cores loads a chunk of a matrix in the cache, it makes an implicit shared cache prefetch to other processors since they will access those elements in cache instead of main memory. Additionally, the superlinear region is wider for parallel execution with greater number of cores, being the infinite (in the analyzed region) for parallel execution with 4 cores.

Both CPUs have different behavior for the efficiency also. Adding more resources on Intel i7 processor increases the speedup, but reduces the efficiency, i.e. the efficiency is greater when using smaller number of cores in parallel execution. The same holds for AMD Opteron but only in the left - sublinear region for smaller matrix size. Despite our hypothesis and Gustafson's Law, in superlinear region (for greater matrix size until the analyzed $N = 1100$) both the speedup and efficiency are saturated, but now adding the more resources will increase the efficiency as well.

VII. Future Work

Many unexpected results are observed: speed, speedup and efficiency discrepant regions, local maximums, speedup peaks, etc which will be the subject of our further research. In this paper we used the "slow" dense MMM without optimization to determine different vendor CPU performance on "slow" cache intensive algorithm. Our future work will be directed towards analyzing specific optimization techniques on different processor architectures, and their contribution of achieving even greater speeds and resolving specific bottlenecks. Moreover, other compute and cache intensive algorithms may be analyzed to achieve better performance on specific CPU cache architectures.

In this paper we determine the region where maximum performance (speed) is achieved and the best scaling (speedup and efficiency) on single chip multicore shared memory multiprocessor. We will try to model the performance on other multiprocessors like multichip multicore or multichip singlecore, also with different cache organization.

References

[1] M. Gusev and S. Ristov, "Matrix multiplication performance analysis in virtualized shared memory multiprocessor," in *MIPRO, 2012 Proceedings of the 35th International Convention, IEEE Conference Publications*, 2012, pp. 264–269.

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

[3] 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.

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

[5] D. J. Lee and T. J. Downar, "The application of posix threads and OpenMP to the U.S. nrc neutron kinetics code parcs," in *Proceedings of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, ser. WOMPAT '01. London, UK, UK: Springer-Verlag, 2001, pp. 90–100.

[6] R. Blikberg and T. Sørevik, "Nested parallelism: Allocation of threads to tasks and OpenMP implementation," *Sci. Program.*, vol. 9, no. 2,3, pp. 185–194, Aug. 2001.

[7] J. Al-Jaroodi, N. Mohamed, H. Jiang, and D. Swanson, "An agent-based infrastructure for parallel java on heterogeneous clusters," in *Proc. of the IEEE Int. Conf. on Cluster Computing*, ser. CLUSTER '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 19–.

[8] M. Kolberg, G. Bohlender, and D. Claudio, "Improving the performance of a verified linear system solver using optimized libraries and parallel computation," in *High Performance Computing for Computational Science - VECPAR 2008*, J. M. Palma and et al., Eds. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 13–26.

[9] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," *J. Supercomp.*, vol. 50, no. 1, pp. 36–77, 2009.

[10] X. Martorell, J. Labarta, N. Navarro, and E. Ayguad, "A library implementation of the nano-threads programming model," in *Euro-Par, Vol. II'96*, 1996, pp. 644–649.

[11] S. Ristov and M. Gusev, "Superlinear speedup for matrix multiplication," in *Information Technology Interfaces, Proceedings of the ITI 2012 34th International Conference on*, 2012, pp. 499–504.

[12] M. Gusev and S. Ristov, "Superlinear speedup in windows azure cloud," in *2012 IEEE 1st International Conference on Cloud Networking (CLOUDNET) (IEEE CloudNet'12)*, Paris, France, Nov 2012, pp. 173–175.

[13] D. P. Playne and K. A. Hawick, "Comparison of gpu architectures for asynchronous communication with finite-differencing applications." *Concurrency and Computation: Practice and Experience*, vol. 24, no. 1, pp. 73–83, 2012.

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

[15] A. M. Castaldo and R. C. Whaley, "Scaling lapack panel operations using parallel cache assignment," *SIGPLAN Not.*, vol. 45, no. 5, pp. 223–232, Jan. 2010.

[16] J. L. Gustafson, "Reevaluating Amdahl's law," *Communication of ACM*, vol. 31, no. 5, pp. 532–533, May 1988.