# Complexity analysis and performance evaluation of matrix product on multicore architectures

Mathias Jacquelin, Loris Marchal and Yves Robert

École Normale Suprieure de Lyon, France
{Mathias.Jacquelin|Loris.Marchal|Yves.Robert}@ens-lyon.fr

## Abstract

*The multicore revolution is underway. Classical algorithms must be revisited in order to take the hierarchical memory layout into account. In this paper, we aim at minimizing the number of cache misses paid during the execution of the matrix product kernel on a multicore processor, and we show how to achieve the best possible tradeoff between shared and distributed caches. Comprehensive simulation results confirm the analytical performance predictions and fully establish the practical significance of our new algorithms.*

## I. Introduction

Dense linear algebra kernels are the key to performance for many scientific applications. Some of these kernels, like matrix multiplication, have extensively been studied on parallel architectures. Two well-known parallel versions are Cannon's algorithm [1] and the ScaLAPACK outer product algorithm [2]. Typically, parallel implementations work well on 2D processor grids: input matrices are sliced horizontally and vertically into square blocks; there is a one-to-one mapping of blocks onto physical resources; several communications can take place in parallel, both horizontally and vertically. Even better, most of these communications can be overlapped with (independent) computations. All these characteristics render the matrix product kernel quite amenable to an efficient parallel implementation on 2D processor grids.

However, algorithms based on a 2D grid (virtual) topology are not well suited for multicore architectures. In particular, in a multicore architecture, memory is shared, and data accesses are performed through a hierarchy of caches, from shared caches to distributed caches. We need to take further advantage of data locality, in order to minimize data movement. This hierarchical framework resembles that of out-of-core algorithms [3] (the shared cache being the disk) and that of master-slave implementations with limited memory [4] (the shared cache being the master's memory). The latter paper [4] presents the Maximum Reuse Algorithm which aims at minimizing the communication volume from the master to the slaves. Here, we adapt this study to multicore architectures, by taking both cache levels into account.

## II. Problem statement

### A. Multicore architectures

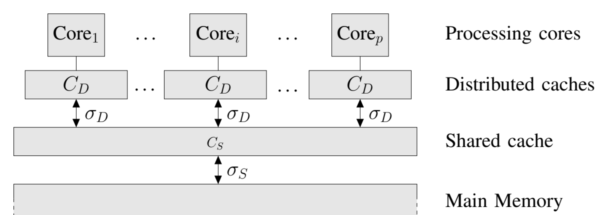



**Figure 1. Multicore architecture model.**

A major difficulty of this study is to come up with a realistic but still tractable model of a multicore processor. We assume that such a processor is composed of $p$ cores, and that each core has the same computing speed. The processor is connected to a memory, which is supposed to be large enough to contain all necessary data (we do not deal with out-of-core execution here). The data path from the memory to a computing core

goes through two levels of caches. The first level of cache is shared among all cores, and has size $C_S$, while the second level of cache is distributed: each core has its own private cache, of size $C_D$. Caches are supposed to be *inclusive*, which means that the shared cache contains *at least* all the data stored in every distributed cache. Therefore, this cache must be larger than the union of all distributed caches: $C_S \geq p \times C_D$. Our caches are also "fully associative", and can therefore store any data from main memory. Figure 1 depicts the multicore architecture model.

The hierarchy of caches is used as follows. When a data is needed in a computing core, it is first sought in the distributed cache of this core. If the data is not present in this cache, a *distributed-cache miss* occurs, and the data is then sought in the shared cache. If it is not present in the shared cache either, then a *shared-cache miss* occurs, and the data is loaded from the memory in the shared cache and afterward in the distributed cache. When a core tries to write to an address that is not in the caches, the same mechanism applies. Rather than trying to model this complex behavior, we assume in the following an *ideal cache model* [5]: we suppose that we are able to totally control the behavior of each cache, and that we can load any data into any cache (shared of distributed), with the constraint that a data has to be first loaded in the shared cache before it could be loaded in the distributed cache. Although somewhat unrealistic, this simplified model has been proven not too far from reality: it is shown in [5] that an algorithm causing $N$ cache misses with an ideal cache of size $L$ will not cause more than $2N$ cache misses with a cache of size $2L$ and implementing a classical LRU replacement policy.

In the following, our objective is twofold: (i) minimize the number of cache misses during the computation of matrix product, and (ii) minimize the predicted data access time of the algorithm. To this end, we need to model the time needed for a data to be loaded in both caches. To get a simple and yet tractable model, we consider that cache speed is characterized by its bandwidth. The shared cache has bandwidth $\sigma_S$, thus a block of size $S$ needs $S/\sigma_S$ time-unit to be loaded from the memory in the shared cache, while each distributed cache has bandwidth $\sigma_D$. Moreover, we assume that concurrent loads to several distributed caches are possible without contention.

Finally, the purpose of the algorithms described below is to compute the classical matrix product $C = A \times B$. In the following, we assume that $A$ has size $m \times z$, $B$ has size $z \times n$, and $C$ has size $m \times n$. We use a block-oriented approach, to harness the power of BLAS routines [2]. Thus, the atomic elements that we manipulate are not matrix coefficients but rather square blocks of coefficients of size $q \times q$. Typically, $q$ ranges from 32 to 100 on most platforms.

## B. Communication volume

The key point to performance in a multicore architecture is efficient data reuse. A simple way to assess data locality is to count and minimize the number of cache misses, that is the number of times each data has to be loaded in a cache. Since we have two types of caches in our model, we try to minimize both the number of misses in the shared cache and the number of misses in the distributed caches. We denote by $M_S$ the number of cache misses in the shared cache. As for distributed caches, since accesses from different caches are concurrent, we denote by $M_D$ the maximum of all distributed caches misses: if $M_D^{(c)}$ is the number of cache misses for the distributed cache of core $c$, $M_D = \max_c M_D^{(c)}$.

In a second step, since the former two objectives are conflicting, we aim at minimizing the overall time $T_{\text{data}}$ required for data movement. With the previously introduced bandwidth, it can be expressed as $T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D}$. Depending on the ratio between cache speeds, this objective provides a tradeoff between both cache miss quantities.

## C. Lower bound on communication

In [3], Irony, Toledo and Tiskin propose a lower bound on the number of communications needed to perform a matrix product. Their study focuses on a system with a memory of size $M$ and concludes that the communication-to-computation ratio of a matrix product is bounded below by $\sqrt{\frac{27}{8M}}$. We have extended this study to our hierarchical cache architecture (details can be found in the corresponding research report [6]). Here, the communication-to-computation is the ratio between the number of cache misses and the number of operations. With *comp*$(c)$ being the amount of computation done by core $c$, we can define this ratio for both types of caches.

For the shared cache, we consider everything above this cache level as a single processor and the main memory as a master which sends and receives data. $CCR_S = M_S / (\sum_c comp(c))$ is therefore the CCR for the shared cache.

In the case of the distributed caches, we first apply the previous result on a single core $c$, with cache size

$C_D$. We thus have $CCR_c \geq \sqrt{\frac{27}{8C_D}}$. We have defined the overall distributed CCR as the average of all $CCR_c$. The overall amount of computation for the matrix product is $mnz$, and in all our algorithms, this amount is equally balanced among cores, so that $comp(c) = mnz/p$ for all cores. Therefore this bound also holds for the $CCR_D$: $CCR_D = \frac{1}{p}\sum_{c=1}^{p}(M_{Dc}/comp(c))$. Indeed, we could even have a stronger result, on the minimum of all $CCR_c$.

In both cases, we have been able to extend the previous bound:

$$CCR_S \geq \sqrt{\frac{27}{8C_S}} \quad \text{and} \quad CCR_D \geq \sqrt{\frac{27}{8C_D}}.$$

# III. Algorithms

In the out-of-core algorithm of [3], the three matrices $A$, $B$ and $C$ are equally accessed throughout time. This naturally leads to allocating one third of the available memory to each matrix. This algorithm has a communication-to-computation ratio of $O\left(\frac{mnz}{\sqrt{M}}\right)$ for a memory of size $M$ but it does not use the memory optimally. The Maximum Reuse Algorithm [4] proposes a more efficient memory allocation: it splits the available memory into $1+\mu+\mu^2$ blocks, storing a square block $C_{i_1\ldots i_2,j_1\ldots j_2}$ of size $\mu^2$ of matrix $C$, a row $B_{i_1\ldots i_2,j}$ of size $\mu$ of matrix $B$ and one element $A_{i,j}$ of matrix $A$ (with $i_1 \leq i \leq i_2$ and $j_1 \leq j \leq j_2$). This allows to compute $C_{i_1\ldots i_2,j_1\ldots j_2} += A_{i,j} \times B_{i_1\ldots i_2,j}$. Then, with the same block of $C$, other computations can be accumulated by considering other elements of $A$ and $B$. The block of $C$ is stored back only when it has been processed entirely, thus avoiding any future need of reading this block to accumulate other contributions. Using this framework, the communication-to-computation ratio is $\frac{2}{\sqrt{M}}$ for large matrices.

To adapt the Maximum Reuse Algorithm to multicore architectures, we must take into account both cache levels. Depending on our objective, we adapt the previous data allocation scheme so as to fit with the shared cache, with the distributed caches, or with both. The main idea is to design a "data-thrifty" algorithm that reuses matrix elements as much as possible and loads each required data only once in a given loop. Since the outermost loop is prevalent, we load the largest possible square block of data in this loop, and adjust the size of the other blocks for the inner loops, according to the objective (shared-cache, distributed-cache, tradeoff) of the algorithm. We define two parameters that will prove helpful to compute the size of the block of $C$ that should be loaded in the shared cache or in a distributed cache:

- $\lambda$ is the largest integer with $1+\lambda+\lambda^2 \leq C_S$;
- $\mu$ is the largest integer with $1+\mu+\mu^2 \leq C_D$.

In the following, we assume that $\lambda$ is a multiple of $\mu$, so that a block of size $\lambda^2$ that fits in the shared cache can be easily divided in blocks of size $\mu^2$ that fit in the distributed caches.

## A. Shared-cache misses

**Algorithm 1**: The algorithm minimizing shared-cache misses.

```
for Step = 1 to (m×n)/λ² do
    Load a new block C_block (of size λ × λ) from C in the shared cache
    for k = 1 to z do
        Load a row B_row (of size λ) from row z of B in the shared cache
        Distribute B_row to the distributed caches
        for l = 1 to λ do
            foreach core c in parallel do
                Load the element a = A[l, k] in the shared and distributed cache
                Load a row C_row (of size λ/p) from C_block in the distributed cache
                Compute the new contribution: C_row ← C_row + a × B_row
                Write back C_row to the shared cache
    Write back the block C_block to main memory
```

To minimize the number of shared-cache misses, we adapt the Maximum Reuse Algorithm with parameter $\lambda$. A square block $C_{\text{block}}$ of size $\lambda^2$ of $C$ is allocated in the shared cache, together with a row of $\lambda$ elements of $B$ and one element of $A$. Then, the row of $C_{\text{block}}$ is distributed and computed by the different cores. This is described in details in Algorithm 1, and the memory layout is depicted in Figure 2.

In this algorithm, the whole matrix $C$ is loaded in the shared cache, thus resulting in $mn$ cache misses. For the computation of each block of size $\lambda^2$, $z$ rows of size $\lambda$ are loaded from $B$, and $z \times \lambda$ elements of $A$ are accessed. Since there are $mn/\lambda^2$ steps, this amounts to a total of $M_S = mn + 2mnz/\lambda$ shared-cache misses. For large matrices, this leads to a shared-cache CCR of $2/\lambda$, which is close to the lower bound.

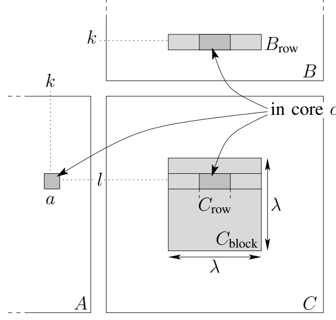



**Figure 2. Data layout for Algorithm 1.**

## B. Distributed-cache misses

Our next objective is to minimize the number of distributed-cache misses. To this end, we use the parameter $\mu$ defined earlier to store in each distributed cache a square block of size $\mu^2$ of $C$, a fraction of row (of size $\mu$) of $B$ and one element of $A$. Contrarily to the previous algorithm, the block of $C$ will be totally computed before being written back to the shared cache. All $p$ cores work on different blocks of $C$. Thanks to the constraint $p \times C_D \leq C_S$, we know that the shared cache has the capacity to store all necessary data. The overall number of distributed cache misses on a core will then be $M_D = \frac{1}{p}(mn + 2mnz/\mu)$ (see [6] for details). For large matrices, this leads to a distributed-cache CCR of $2/\mu$, which is close to the lower bound.

## C. Data access time

To get a tradeoff between minimizing the number of shared-cache and distributed-cache misses, we now aim at minimizing $T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D}$. The sketch of the algorithm, detailed in [6], is the following:

1) A block from $C$ of size $\alpha \times \alpha$ is loaded in the shared cache. Its size satisfies $p \times \mu^2 \leq \alpha^2 \leq \lambda^2$. Both extreme cases are obtained when one of $\sigma_D$ and $\sigma_S$ is negligible in front of the other.
2) In the shared cache, we also load a block from $B$, of size $\beta \times \alpha$, and a block from $A$ of size $\alpha \times \beta$. Thus, we have $2\alpha \times \beta + \alpha^2 \leq C_D$.
3) The $\alpha \times \alpha$ block of $C$ is split into sub-blocks of size $\mu \times \mu$ which are processed by the different cores. These sub-blocks of $C$ are cyclicly distributed among every distributed-caches. The same holds for the block-row of $B$ which is split into $\beta \times \mu$ block-rows and cyclically distributed, row by row (i.e. by blocks of size $1 \times \mu$), among every distributed-cache.
4) The contribution of the corresponding $\beta$ (fractions of) columns of $A$ and $\beta$ (fractions of) lines of $B$ is added to the block of $C$. Then, another $\mu \times \mu$ block of $C$ residing in shared cache is distributed among every distributed-cache, going back to step 3.
5) As soon as all elements of $A$ and $B$ have contributed to the $\alpha \times \alpha$ block of $C$, another $\beta$ columns/lines from $A/B$ are loaded in shared cache, going back to step 2.
6) Once the $\alpha \times \alpha$ block of $C$ in shared cache is totally computed, a new one is loaded, going back to step 1.

With this algorithm, we get: $T_{\text{data}} = \frac{1}{\sigma_S}(mn + \frac{2mnz}{\alpha}) + \frac{1}{\sigma_D}(\frac{mnz}{p\beta} + \frac{2mnz}{p\mu})$. Together with the constraint $2\alpha \times \beta + \alpha^2 \leq C_D$, it allows to compute the best value for parameters $\alpha$ and $\beta$, depending on the ratio $\sigma_S/\sigma_D$ (see [6] for details).

# IV. Simulation results

We have presented three algorithms minimizing different objectives (shared cache misses, distributed cache misses and overall time spent in data movement) and provided a theoretical analysis of their performance. However, our simplified multicore model makes some assumptions that are not realistic on a real hardware platform. In particular it uses an ideal and omniscient data replacement policy instead of a classical LRU policy. This led us to design a multicore cache simulator and implement all our algorithms, as well as the outer-product [2] and Toledo [3] algorithms, using different cache policies. The goal is to experimentally assess the impact of the policies on the actual performance of the algorithms, and to measure the gap between the theoretical prediction and the observed behavior. The main motivation behind the choice of a simulator instead of a real hardware platform resides in commodity reasons: simulation enables to obtain desired results faster and allows to easily modify multicore processor parameters (cache sizes, number of cores, bandwidths, . . . ).

## A. Settings

The driving feature of our simulator was simplicity. It implements the cache hierarchy of our model, and basically counts the number of cache misses in each cache level. It offers two data replacement policies, *LRU* (Least Recently Used) and *Ideal*. In the LRU

mode, read and write operations are made at the distributed cache level (top of hierarchy); if a miss occurs, operations are propagated throughout the hierarchy until a cache hit happens. In the Ideal mode, the user manually decides which data needs to be loaded/unloaded in a given cache; I/O operations are not propagated throughout the hierarchy in case of a cache miss: it is the user responsibility to guarantee that a given data is present in every caches below the target cache.

We have implemented two reference algorithms: (i) *Outer Product*, the algorithm in [2], for which we organize cores as a (virtual) processor torus and distribute square blocks of data elements to be updated among them; and (ii) *Equal*, an algorithm inspired by [3], which uses a simple equal-size memory scheme: one third of distributed caches is equally allocated to each loaded matrix sub-block. In fact, the algorithm in [3] deals with a single cache level, hence we decline it in two versions, *Shared Equal* for shared cache optimization, and *Distributed Equal* for distributed cache optimization. We have also implemented the three versions of the Multicore Maximum Reuse Algorithm:

- *Shared Opt.*, the version to minimize the number of shared caches misses $M_S$
- *Distributed Opt.*, the version to minimize the number of distributed cache misses $M_D$
- *Tradeoff*, the version to minimize the data access time $T_{\text{data}}$.

In the experiments, we simulated a "realistic" quad-core processor with 8MB of shared cache and four distributed caches of size 256KB dedicated to both data and instruction. We assume that two-thirds of the distributed caches are dedicated to data, and one-third for the instructions. See [6] for results using a more pessimistic repartition of one half for data and one-half for instructions. Recall that square blocks of matrix coefficients have size $q \times q$. We report results for $q = 32$, and we derive $C_S = 977$ and $C_D = 21$. See [6] for results with larger blocks ($q = 64$, $C_S = 245$ & $C_D = 6$, and $q = 80$, $C_S = 157$ & $C_D = 4$).

## B. LRU vs IDEAL

Here we assess the impact of the data replacement policy on the number of shared cache misses and on the performance achieved by the algorithm. Figure 3 shows the total number of shared cache misses for Shared Opt., in function of the matrix dimension. While $LRU(C_S)$ (the LRU policy with a cache of size $C_S$) achieves significantly more cache-misses than predicted by the theoretical formula, $LRU(2C_S)$ is quite close, thereby experimentally validating the prediction of [5]. Similar results are obtained for Shared Equal. Furthermore, the same conclusions hold for Distributed Opt. and Distributed Equal, see [6]. Note that Outer Product is insensitive to cache policies.

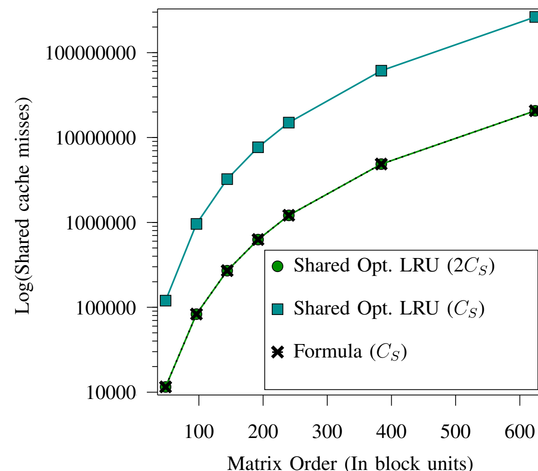



**Figure 3. Impact of LRU policy on the number of shared cache misses $M_S$ with $C_S = 977$.**

This leads us to run our tests using the following two simulation settings:

- The **IDEAL** setting, which corresponds to the use of the omniscient ideal data replacement policy assumed in the theoretical model. It relies on the Ideal mode of the simulator and uses entire cache sizes ($C_S$ and/or $C_D$) as a parameter for the algorithms
- The **LRU-50%** setting, which relies on a LRU cache data replacement policy, but uses only one half of cache sizes as a parameter for the algorithms. The other half is used by the LRU policy as kind of an automatic prefetching buffer.

## C. Performance

*1) Shared Opt.:* Figure 4(a) depicts the number of shared cache misses achieved by Shared Opt versions LRU-50% and IDEAL, in comparison with Outer Product, Shared Equal and the lower bound $m^3\sqrt{\frac{27}{8C_S}}$, according to the matrix dimension $m$. We see that Shared Opt. performs significantly better than Outer Product and Shared Equal for the LRU-50% policy. Under the IDEAL policy, it is closer to the lower bound, but this latter setting is not realistic.

*2) Distributed Opt.:* Figure 4(b) is the counterpart of Figure 4(a) for distributed caches. Similarly, we see that Distributed Opt. performs significantly better than Outer Product and Distributed Equal for the LRU-50% policy. Under the IDEAL policy, it is very close to the lower bound $\frac{m^3}{p}\sqrt{\frac{27}{8C_D}}$.

*3) Tradeoff:* The overall time spent in data movement $T_{\text{data}}$ of all six LRU-50% algorithms is shown on Figure 4(c). Here, we observe an unexpected tie: Tradeoff algorithm does not rank better than Shared Opt. This trend is probably due to the artificial constraints set on cache-related parameters: for instance we require that $\alpha$ divides $m$ and is a multiple of $\sqrt{p}$ and of $\mu$.

In the implementations, parameter $\lambda$ and $\alpha$ can be significantly lower than their optimal numerical value. Nevertheless, looking at Figure 4(d), we see that Tradeoff does outperform other algorithms with the IDEAL policy. We also run another experiment to assess the impact of cache bandwidths on $T_{\text{data}}$. We introduce the parameter $r$ defined as: $r = \frac{\sigma_S}{\sigma_D+\sigma_S}$. Figure 5 reports results for square matrices of size $m = 240$.

We see that Tradeoff performs best, and still offers the best performance even after distributed misses have become predominant. When the latter event occurs, plots cross over: Shared Opt. and Distributed Opt. achieve the same $T_{\text{data}}$. We also point out that when $r = 0$, Tradeoff achieves almost the same $T_{\text{data}}$ than Shared Opt., while when $r = 1$, it ties Distributed Opt.

## V. Related work

**Algorithms–** In [3], the authors introduce several lower bounds on the communication volume for standard matrix multiplication algorithms. The scope of their work ranges from one processor and its main memory to several distributed memory processors. They also provide a lower bound for a processor having a fast cache and a large slow memory. In [4], the authors introduce the Maximum Reuse Algorithm, a matrix product algorithm for master-slave platforms. They improve the lower bound introduced in [3], and show that their algorithm is close to this bound for large matrices. However, neither [3] nor [4] deals with multicore processors and the additional level of cache hierarchy that they imply.

**Models–** The *ideal-cache* model is presented in [5], together with the cache-oblivious paradigm, which aims at provide asymptotically optimal "cache-unaware" algorithms. A key contribution is the proof that the ideal cache-model can efficiently be simulated with a LRU replacement policy. However, the model only focuses on single-core processors. It is extended in [7] for multicore processors: the authors of [7] study divide-and-conquer cache-oblivious algorithms for several problems,

and they design algorithms that are asymptotically optimal for both shared and distributed caches misses. The emphasis is on models and asymptotic performance rather than on algorithms for fixed-size matrices.

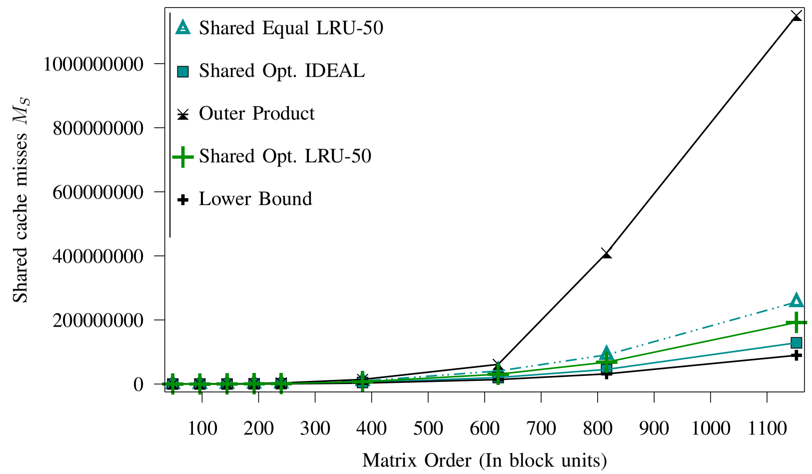
In [8], the authors introduce a theoretical model for multicore processors intended to be used to analyze the complexity of algorithms on these new platforms. They also describe a framework called SWARM that aims at providing an open-source library for developing software on multicore architectures. However, they do not explicitly use the notion of cache misses, but instead focus on the number of blocks transferred between shared cache and main memory; distributed caches are not considered in their analysis.

In [9], the authors study the behavior of DGEMM kernels and introduce a fine-tuning version leading to better performance than Intel's parallel DGEMM in the MKL library. This paper provides a fine-grain analysis in terms of cache related notions, as for instance cache misses or false sharing, on a real hardware platform. Our analysis is at a higher level (our algorithms call sequential DGEMM kernels) and is not associated to any particular hardware architecture.

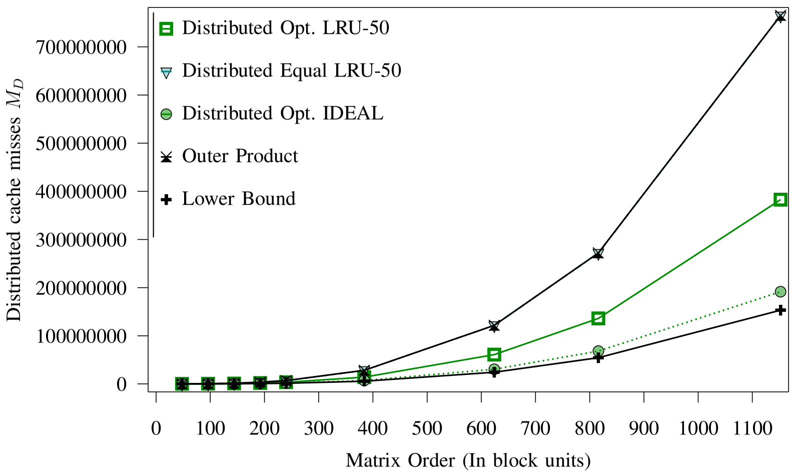
**Experiments–** In [10], the authors present performance results for dense linear algebra using recent NVIDIA GPUs, and analyse some factors impacting performance on these particular multicore processors through the performance evaluation of their matrix-matrix multiplication kernel. This work also is at a lower level since it focuses on implementing a better DGEMM routine for GPUs.

## VI. Conclusion

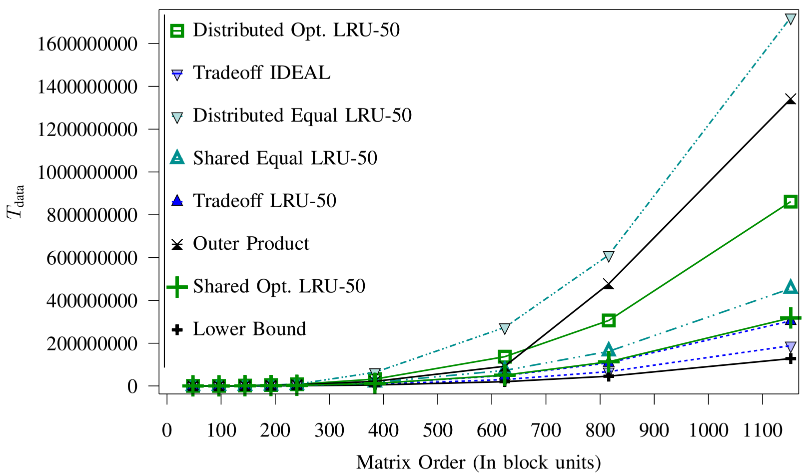
In this paper, we proposed cache-aware algorithms for multicore processors. We have proposed a model for multicore memory layout. Using this model, we have extended a lower bound on cache misses, and proposed cache-aware algorithms. For both types of caches, our algorithms reach a CCR which is close to the lower bound for large matrices. We also propose an algorithm for minimizing the overall data access time, which realizes a tradeoff between shared and distributed cache misses. Every algorithm introduced in this paper has been tested, and is behavior validated, on the simulator that we have designed, using realistic parameters.



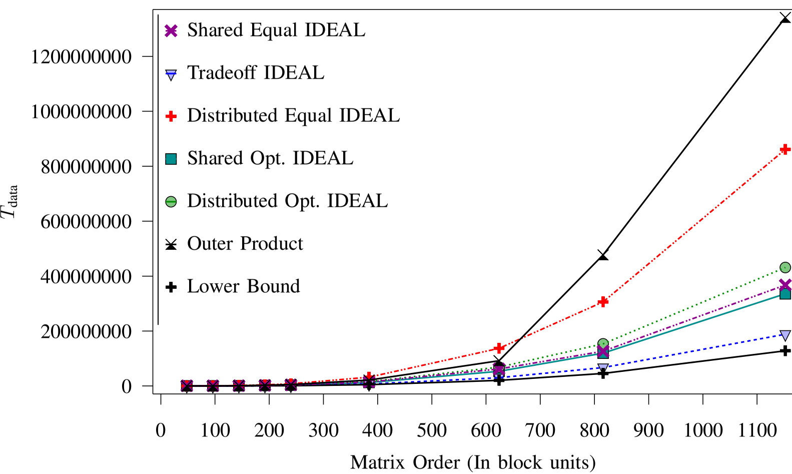

(a) Shared cache misses $M_S$ in function of matrix order.




(b) Distributed cache misses $M_D$.




(c) Overall data time $T_{\text{data}}$ (LRU policy).




(d) Overall data time $T_{\text{data}}$ (IDEAL policy).

**Figure 4. Simulation Results**

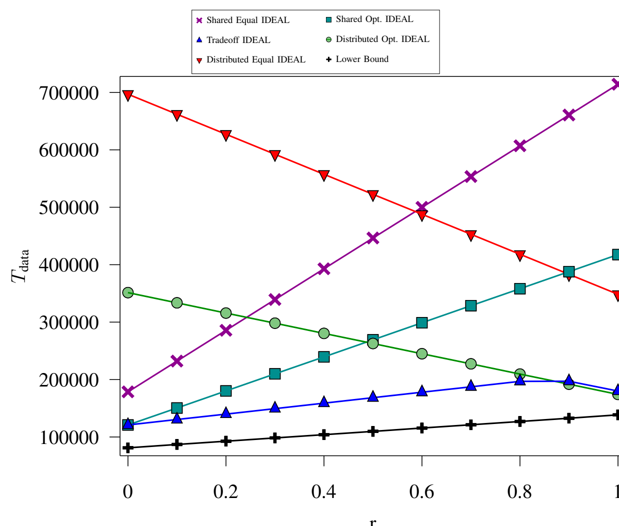



**Figure 5. Cache bandwidth impact on $T_{\text{data}}$ in function of $r$, the ratio between $\sigma_S$ and $\sigma_D$. Unit blocks of size $q = 32$, results are given for a square matrix of dimension $m = 240$. $C_S = 977$ and $C_D = 21$ (data occupy two thirds of distributed caches).**

Future work will be twofold. On the algorithmic side, we will tackle more complex operations, such as LU factorization or path problems. On the more practical side, we will implement all algorithms on state-of-the-art multicore machines. This will be a first step towards the (more ambitious) task of designing efficient algorithms for clusters of multicores: we expect yet another level of hierarchy (or tiling) in the algorithmic specification to be required in order to match the additional complexity of such platforms.

## References


[1] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, Montana State University, 1969.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. SIAM, 1997.

[3] S. Toledo, "A survey of out-of-core algorithms in numerical linear algebra," in *External Memory Algorithms and Visualization*. American Mathematical Society Press, 1999, pp. 161–180.

[4] J.-F. Pineau, Y. Robert, F. Vivien, and J. Dongarra, "Matrix product on heterogeneous master-worker platforms," in *PPoPP'2008, the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, 2008, pp. 53–62.

[5] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *FOCS'99, the 40th IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 1999, pp. 285–298.

[6] M. Jacquelin, L. Marchal, and Y. Robert, "Complexity analysis and performance evaluation of matrix product on multicore architectures," LIP, ENS Lyon, Research report RRLIP2009-09, 2009. [Online]. Available: http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2009/RR2009-09.pdf

[7] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, "Provably good multicore cache performance for divide-and-conquer algorithms," in *SODA'08, the 19th ACM-SIAM symposium on Discrete algorithms*. SIAM Press, 2008, pp. 501–510.

[8] D. A. Bader, V. Kanade, and K. Madduri, "SWARM: A parallel programming framework for multicore processors," in *IPDPS'07, the 21st IEEE Int. Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.

[9] S. Zuckerman, M. Pérache, and W. Jalby, "Fine tuning matrix multiplications on multicore," in *High Pefroamnce Computing HiPC'08*. Springer Verlag LNCS 5374, 2008, pp. 30–41.

[10] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *SC'08, the 2008 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 2008, pp. 1–11.