The impact of cache misses on the performance of matrix product algorithms on multicore platforms

Mathias Jacquelin, Loris Marchal, Yves Robert

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The impact of cache misses on the performance of matrix product algorithms on multicore platforms

Mathias Jacquelin — Loris Marchal — Yves Robert

N° 7456

Novembre 2010

Distributed and High Performance Computing

ISSN 0249-6399 ISRN INRIA/RR--7456--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

The impact of cache misses on the performance of matrix product algorithms on multicore platforms

Mathias Jacquelin* , Loris Marchal† , Yves Robert*‡

Theme : Distributed and High Performance Computing
Équipe-Projet GRAAL

Rapport de recherche n° 7456 — Novembre 2010 — 29 pages

Abstract: The multicore revolution is underway, bringing new chips introducing more complex memory architectures. Classical algorithms must be revisited in order to take the hierarchical memory layout into account. In this paper, we aim at designing cache-aware algorithms that minimize the number of cache misses paid during the execution of the matrix product kernel on a multicore processor. We analytically show how to achieve the best possible tradeoff between shared and distributed caches. We implement and evaluate several algorithms on two multicore platforms, one equipped with one Xeon quadcore, and the second one enriched with a GPU. It turns out that the impact of cache misses is very different across both platforms, and we identify what are the main design parameters that lead to peak performance for each target hardware configuration.

Key-words: Multicore platform, Matrix product, Cache misses, Cache-aware algorithms.

* École Normale Supérieure de Lyon
† CNRS
‡ Institut Universitaire de France

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Impact des défauts de cache sur les performances des algorithmes de produit de matrices pour les plates-formes multi-cœur

Résumé : La révolution multi-cœur est en cours, qui voit l'arrivée de processeurs dotées d'une architecture mémoire complexe. Les algorithmes les plus classiques doivent être revisités pour prendre en compte la disposition hiérarchique de la mémoire. Dans ce rapport, nous étudions des algorithmes prenant en compte les caches de données qui minimisent le nombre de défauts de cache pendant l'exécution d'un produit de matrices sur un processeur multi-cœur. Nous montrons analytiquement comment obtenir le meilleur compromis entre les caches partagés et distribués. Nous proposons une implémentation pour évaluer ces algorithmes sur deux plates-formes multi-cœur, l'une équipé d'un processeur Xeon quadri-cœur, l'autre dotée d'un GPU. Il apparaît que l'impact des défauts de cache est très différent sur ces deux plates-formes, et nous identifions quels sont les principaux paramètres de conception qui conduisent aux performances maximales pour chacune de ces configurations matérielles.

Mots-clés : plate-forme multi-cœur, produit de matrice, défaut de cache, algorithmes conscient du cache.

1 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. To increase performance, 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 extend this study to multicore architectures, by taking both the shared and distributed cache levels into account. We analytically show how to achieve the best possible tradeoff between shared and distributed caches.

We implement and evaluate several algorithms on two multicore platforms, one equipped with one Xeon quadcore, and the second one enriched with a GPU. It turns out that the impact of cache misses is very different across both platforms, and we identify what are the main design parameters that lead to peak performance for each target hardware configuration. For the sake of reusability, the source code for the implementation and comparison of all algorithms is publicly available at `http://graal.ens-lyon.fr/~mjacquel/mmre_cpu_gpu.html`.

The rest of this paper is organized as follows. Section 2 introduces the model for multicore platforms, and derives new bounds on the number of shared and distributed cache misses of any matrix product algorithm. These bounds derive from a refined analysis of the CCR (communication to computation ratio) imposed by the underlying architecture. Section 3 presents the new algorithms designed to optimize shared cache misses, distributed cache misses, or a combination of both. In Section 4 we proceed to an experimental evaluation of these algorithms, together with a bunch of reference algorithms and with the vendor library routines, on a CPU platform. Section 5 is the counterpart for a GPU platform. Finally in Section 6, we provide final remarks, and directions for future work.

2 Problem statement

2.1 Multicore architectures

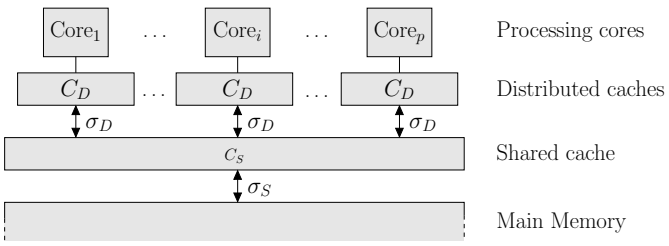


Figure 1: Multicore architecture model.

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 $2L$ 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 σ_S, thus a block of size S needs S/σ_S time-unit to be loaded from the memory in the shared cache, while each distributed cache has bandwidth σ_D. Moreover, we assume that concurrent loads to several distributed caches are possible without contention. Since we assume an ideal cache model with total control on the cache behavior, we also suppose that coherency mechanisms' impact on performance can be neglected.

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.

2.2 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_{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.

2.3 Lower bound on communications

In [3], Irony, Toledo and Tiskin propose a lower bound on the number of communications needed to perform a matrix product. We have extended this study to the hierarchical cache architecture. In what follows, we consider a computing system (which consists of one or several computing cores) using a cache of size Z. We estimate the number of computations that can be performed owing to Z consecutive cache misses, that is owing to Z consecutive load operations. We recall that each matrix element is in fact a matrix block of size $q \times q$. We use the following notations :

- Let η_{old}, ν_{old}, and ξ_{old} be the number of blocks in the cache used by blocks of A, B and C just before the Z cache misses.
- Let η_{read}, ν_{read}, and ξ_{read} be the number of blocks of A, B and C read in the main memory when these Z cache misses occurs.
- Let $comp(c)$ be the amount of computation done by core c

Before the Z cache misses, the cache holds at most Z blocks of data, therefore, after Z cache misses, we have:

$$\begin{cases} \eta_{old} + \nu_{old} + \xi_{old} \leq Z \\ \eta_{read} + \nu_{read} + \xi_{read} = Z \end{cases} \tag{1}$$

2.3.1 Loomis-Whitney's inequality

The following lemma, given in [6] and based on Loomis-Whitney inequality, is valid for any conventional matrix multiplication algorithm $C = AB$, where A is $m \times z$, B is $z \times n$ and C is $m \times n$. A processor that contributes to N_C elements of C and accesses N_A elements of A and N_B elements of B can perform at most $\sqrt{N_A N_B N_C}$ elementary multiplications. According to this lemma, if we let K denote the number of elementary multiplications performed by the computing system, we have:

$$K \leq \sqrt{N_A N_B N_C}$$

No more than $(\eta_{old} + \eta_{read})q^2$ elements of A are accessed, hence $N_A = (\eta_{old} + \eta_{read})q^2$. The same holds for B and C: $N_B = (\nu_{old} + \nu_{read})q^2$ and $N_C = (\xi_{old} + \xi_{read})q^2$. Let us simplify the notations using the following variables:

$$\begin{cases} \eta_{old} + \eta_{read} = \eta \times Z \\ \nu_{old} + \nu_{read} = \nu \times Z \\ \xi_{old} + \xi_{read} = \xi \times Z \end{cases} \tag{2}$$

Then we derive $K \leq \sqrt{\eta\nu\xi} \times Z\sqrt{Z} \times q^3$. Writing $K = kZ\sqrt{Z}q^3$, we obtain the following system of equations:

$$\text{MAXIMIZE } k \text{ SUCH THAT}$$
$$\begin{cases} k \leq \sqrt{\eta\nu\xi} \\ \eta + \nu + \xi \leq 2 \end{cases}$$

Note that the second inequality comes from Equation (1). This system admits a solution which is $\eta = \nu = \xi = \frac{2}{3}$ and $k = \sqrt{\frac{8}{27}}$. This gives us a lower bound on the communication-to-computation ratio (in terms of blocks) for any matrix multiplication algorithm:

$$CCR \geq \frac{Z}{k \times Z\sqrt{Z}} = \sqrt{\frac{27}{8Z}}$$

2.3.2 Bound on shared-cache misses

We will first use the previously obtained lower bound to study shared-cache misses, considering everything above this cache level as a single processor and the main memory as a master which sends and receives data. Therefore, with $Z = C_S$ and $K = \sum_c comp(c)$, we have a lower bound on the communication-to-computation ratio for shared-cache misses:

$$CCR_S = \frac{M_S}{K} = \frac{M_S}{\sum_c comp(c)} \geq \sqrt{\frac{27}{8C_S}}$$

2.3.3 Bound on distributed-caches misses

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 define the overall distributed CCR as the average of all CCR_c, so this result also holds for the CCR_D:

$$CCR_D = \frac{1}{p}\sum_{c=1}^{p}\left(\frac{M_{Dc}}{comp(c)}\right) \geq \sqrt{\frac{27}{8C_D}}$$

Indeed, we could even have a stronger result, on the minimum of all CCR_c.

2.3.4 Bound on overall data access time

The previous bound on the CCR can be extended to the data access time, as it is defined as a linear combination of both M_S and M_D:

$$T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D}$$

We can bound M_S using the bound on the CCR:

$$M_S = CCR_S \times mnz \geq mnz \times \sqrt{\frac{27}{8C_S}}$$

As for distributed-cache misses, it is more complex, since M_D is the maximum of all misses on distributed caches, and CCR_D is the average CCR among all cores. In order to get a tight bound, we consider only algorithms where both computation and cache misses are equally distributed among all cores (this applies to all algorithms developed in this paper). In this case, we have

$$M_D = \max_c M_D^{(c)} = M_D^{(0)} = \frac{mnz}{p} \times CCR_0 \geq \frac{mnz}{p} \times \sqrt{\frac{27}{8C_D}}$$

Thus, we get the following bound on the overall data access time:

$$T_{\text{data}} \geq mnz \times \left(\frac{1}{\sigma_S} \times \sqrt{\frac{27}{8C_S}} + \frac{1}{p\sigma_D} \times \sqrt{\frac{27}{8C_D}}\right).$$

3 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 μ^2 of matrix C, a row $B_{i_1 \ldots i_2, j}$ of size μ 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 extend the Maximum Reuse Algorithm to multicore architectures, we must take into account both cache levels. Depending on the objective, we modify the previous data allocation scheme so as to fit with the shared cache, with the distributed caches, or with both. This will lead to three versions of a new multicore Maximum Reuse Algorithm, or MMRA: SharedMMRA, DistributedMMRA and TradeoffMMRA. In all cases, 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:

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

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

3.1 Minimizing shared-cache misses

Algorithm 1: SharedMMRA.

for Step $= 1$ *to* $\frac{m \times n}{\lambda^2}$ **do**
  Load a new block $C[i, \ldots, i+\lambda;\ j, \ldots, j+\lambda]$ of C in the shared cache
  for $k = 1$ *to* z **do**
    Load a row $B[k;\ j, \ldots, j+\lambda]$ of B in the shared cache
    for $i' = i$ *to* $i + \lambda$ **do**
      Load the element $a = A[k;\ i']$ in the shared cache
      foreach *core* $c = 1 \ldots p$ *in parallel* **do**
        Load the element $a = A[k;\ i']$ in the distributed cache of core c
        for $j' = j + (c-1) \times \frac{\lambda}{p}$ *to* $j + c \times \frac{\lambda}{p}$ **do**
          Load $B_c = B[k;\ j']$ in the distributed cache of core c
          Load $C_c = C[i';\ j']$ in the distributed cache of core c
          Compute the new contribution: $C_c \leftarrow C_c + a \times B_c$
          Update block C_c in the shared cache
  Write back the block of C to the main memory

To minimize the number of shared-cache misses M_S, we extend the Maximum Reuse Algorithm with the new parameter λ. A square block C_{block} of size λ^2 of C is allocated in the shared cache, together with a row of λ elements of B and one element of A. Then, each row of C_{block} is divided into sub-rows of $\frac{\lambda}{p}$ elements, which are then distributed to the computing cores together with corresponding element of B and one element of A, updated by the different cores, and written back in shared cache. As soon as C_{block} is completely updated, it is written back in main memory and a new C_{block} is loaded. This is described in Algorithm 1, and the memory layout is depicted in Figure 2.

Note that the space required in each distributed-cache to process one block of q^2 elements of C is 1+1+1 blocks. For now, we have not made any assumption on the size of distributed caches. Let

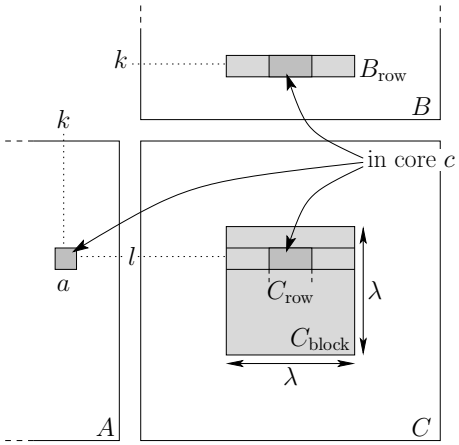


Figure 2: Data layout for Algorithm 1.

S_D be the size of each distributed cache, expressed in the number of matrix coefficients (remember that C_D is expressed in blocks): the constraint $3 \leq C_D$ simply translates into $3q^2 \leq S_D$. We compute the number of cache misses of SharedMMRA as follows:

- *Shared-cache misses*

  In the 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 λ^2, z rows of size λ are loaded from B, and $z \times \lambda$ elements of A are accessed. Since there are mn/λ^2 steps, this amounts to a total number of shared cache misses of:

  $$M_S = \frac{mn}{\lambda^2} \times z \times \left(\lambda + \lambda + \lambda^2\right) = mn + \frac{2mnz}{\lambda}$$

  The communication-to-computation ratio is therefore:

  $$CCR_S = \frac{mn + \frac{2mnz}{\lambda}}{mnz} = \frac{1}{z} + \frac{2}{\lambda}$$

  For large matrices, this leads to a shared-cache CCR of $2/\lambda$, which is close to the lower bound derived earlier.

- *Distributed-caches misses*

  In the algorithm, each block of C is sequentially updated z times by rows of λ elements, each row is distributed element per element, thus requiring $\frac{\lambda}{p}$ steps. Therefore, each distributed cache holds one element of C. This results in $\frac{mnz}{p}$ cache misses. For the computation of each block of size λ^2, $z \times \frac{\lambda}{p}$ elements are loaded from B in each distributed cache, and $z \times \lambda$ elements of A are accessed. Since there are mn/λ^2 steps, this amounts to a total number of:

  $$M_D = \frac{\frac{mn}{\lambda^2} \times z \times \lambda \times p \times \left(1 + \frac{\lambda}{p} + \frac{\lambda}{p}\right)}{p} = \frac{2mnz}{p} + \frac{mnz}{\lambda}$$

  The communication-to-computation ratio is therefore:

  $$CCR_D = \frac{\frac{2mnz}{p} + \frac{mnz}{\lambda}}{\frac{mnz}{p}} = 2 + \frac{p}{\lambda}$$

  This communication-to-computation does not depend upon the dimensions of the matrices, and is the same for large matrices. Moreover, it is far from the lower bound on distributed cache misses.

Algorithm 2: DISTRIBUTEDMMRA.

$offset_i = (\text{My\_Core\_Num}() - 1) \pmod{\sqrt{p}}$
$offset_j = \lfloor \frac{\text{My\_Core\_Num}()-1}{\sqrt{p}} \rfloor$
for Step $= 1$ *to* $\frac{m \times n}{p\mu^2}$ **do**
  Load a new block $C[i, \ldots, i + \sqrt{p}\mu;\ j, \ldots, j + \sqrt{p}\mu]$ of C in the shared cache
  foreach *core* $c = 1 \ldots p$ *in parallel* **do**
    Load
    $C_c = C[i + offset_i \times \mu, \ldots, i + (offset_i + 1) \times \mu;\ j + offset_j \times \mu, \ldots, j + (offset_j + 1) \times \mu]$
    in the distributed cache of core c
  for $k = 1$ *to* z **do**
    Load a row $B[k;\ j, \ldots, j + \sqrt{p}\mu]$ of B in the shared cache
    foreach *core* $c = 1 \ldots p$ *in parallel* **do**
      Load $B_c = B[k;\ j + offset_j \times \mu, \ldots, j + (offset_j + 1) \times \mu]$ in the distributed cache of core c
      for $i' = i + offset_i \times \mu$ *to* $i + (offset_i + 1) \times \mu$ **do**
        Load the element $a = A[k;\ i']$ in the shared cache
        Load the element $a = A[k;\ i']$ in the distributed cache of core c
        Compute the new contribution: $C_c \leftarrow C_c + a \times B_c$
  foreach *core* $c = 1 \ldots p$ *in parallel* **do**
    Update block C_c in the shared cache
  Write back the block of C to the main memory

3.2 Minimizing distributed-cache misses

The next objective is to minimize the number of distributed-cache misses. To this end, we use the parameter μ defined earlier to store in each distributed cache a square block of size μ^2 of C, a fraction of row (of size μ) 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 $\mu \times \mu$ blocks of C are distributed among the distributed caches in a 2-D cyclic way, because it helps reduce (and balance between A and B) the number of shared-cache misses: in this case, assuming that $\sqrt{p}$ is an integer, we load a $\sqrt{p}\mu \times \sqrt{p}\mu$ block of C in shared cache, together with a row of $\sqrt{p}\mu$ elements of B. Then, $\sqrt{p} \times \mu$ elements from a column of A are sequentially loaded in the shared cache ($\sqrt{p}$ non contiguous elements are loaded at each step), then distributed among distributed caches (cores in the same "row" (resp. "column") accumulate the contribution of the same (resp. different) element of A but of different (resp. the same) $\sqrt{p} \times \mu$ fraction of row from B). We compute the number of cache misses of DISTRIBUTEDMMRA (see Algorithm 2) as follows:

- *Shared-cache misses*

  The number of shared-cache misses is:

$$M_S = \frac{mn}{p\mu^2} \times \left(p\mu^2 + z \times 2\sqrt{p}\mu\right) = mn + \frac{2mnz}{\mu\sqrt{p}}$$

  Hence, the communication-to-computation ratio is:

$$CCR_S = \frac{mn + \frac{2mnz}{\mu\sqrt{p}}}{mnz} = \frac{1}{z} + \frac{2}{\mu\sqrt{p}}$$

For large matrices, the communication-to-computation ratio is $\frac{2}{\mu\sqrt{p}} = \sqrt{\frac{32}{8\times\sqrt{p}C_D}}$. This is far from the lower bound since $\sqrt{p}C_D \leq pC_D \leq C_S$.

- *Distributed-caches misses*

  The number of distributed-cache misses is:

$$M_D = \frac{\frac{mn}{p\mu^2} \times p \times \left(\mu^2 + z \times 2\mu\right)}{p} = \frac{mn}{p} + \frac{2mnz}{\mu \times p}$$

  Therefore, the communication-to-computation ratio is:

$$CCR_D = \frac{\frac{mn}{p} + \frac{2mnz}{\mu\times p}}{\frac{mnz}{p}} = \frac{1}{z} + \frac{2}{\mu}$$

  For large matrices, the communication-to-computation ratio is asymptotically close to the value $\frac{2}{\mu} = \sqrt{\frac{32}{8C_D}}$, which is close to the lower bound $\sqrt{\frac{27}{8C_D}}$.

3.3 Minimizing data access time

The two previous objectives are antagonistic: for both previous algorithms, optimizing the number of cache misses of one type leads to a large number of cache misses of the other type. Indeed, minimizing M_S ends up with a number of distributed-cache misses proportional to the common dimension of matrices A and B, and in the case of large matrices this is clearly problematic. On the other hand, focusing only on M_D is not efficient since we dramatically under-use the shared cache: a large part of it is not utilized.

This motivates us to look for a tradeoff between the latter two solutions. However, both kinds of cache misses have different costs, since bandwidths between each level of the memory architecture are different. Hence we introduce the overall time for data movement, defined 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 numbers. To derive an algorithm optimizing this tradeoff, we start from the algorithm presented for optimizing the shared-cache misses. Looking closer to the downside of this algorithm, which is the fact that the part of M_D due to the elements of C is proportional to the common dimension z of matrices A and B, we see that we can reduce this amount by loading blocks of β columns (resp. of rows) of A (resp. B). This way, square blocks of C will be processed longer by the cores before being unloaded and written back in shared-cache, instead of being unloaded after that every element of the column of A residing in shared-cache has been used. However, blocks of C must be smaller than before, and instead of being λ^2 blocks, they are now of size α^2 where α and β are defined under the constraint $2\alpha \times \beta + \alpha^2 \leq C_D$. The sketch of the TradeoffMMRA (see Algorithm 3) is the following:

1. A block of size $\alpha \times \alpha$ of C 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 σ_D and σ_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 cyclicly distributed, row by row (i.e. by blocks of size $1 \times \mu$), among every distributed-caches.
4. The contribution of the corresponding β (fractions of) columns of A and β (fractions of) lines of B is added to the block of C. Then, another $\mu \times \mu$ block of C residing in the shared cache is distributed among every distributed-caches, 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 β 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.

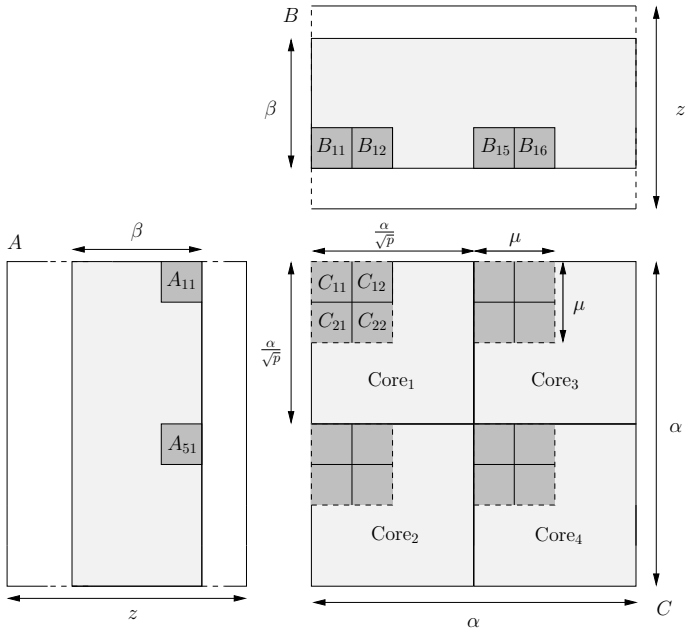


Figure 3: Data distribution of matrices A, B and C: light gray blocks reside in shared-cache, dark gray blocks are distributed among distributed-caches ($\alpha = 8, \mu = 2,\ p = 4$).

We compute the number of cache misses as follows:

- *Shared-cache misses*

  The number of shared-cache misses is given by:

  $$M_S = \frac{mn}{\alpha^2}\left(\alpha^2 + \frac{z}{\beta} \times 2\alpha\beta\right) = mn + \frac{2mnz}{\alpha}$$

  The communication-to-computation ratio is therefore:

  $$CCR_S = \frac{1}{z} + \frac{2}{\alpha}$$

  For large matrices, the communication-to-computation ratio is asymptotically close to $\frac{1}{\alpha}\sqrt{\frac{32}{8}}$ which is farther from the lower bound than the shared-cache optimized version since $\alpha \leq \lambda \approx \sqrt{C_S}$.

- *Distributed-caches misses*

  In the general case (i.e. $\alpha > \sqrt{p}\mu$), the number of distributed-cache misses achieved by TradeoffMMRA is:

  $$M_D = \frac{1}{p} \times \frac{mn}{\alpha^2} \times \left[\frac{\alpha^2}{\mu^2} \times \left(\mu^2 \times \frac{z}{\beta} + \frac{z}{\beta} \times 2\beta\mu\right)\right] = \frac{mn}{p} \times \frac{z}{\beta} + \frac{2mnz}{p\mu}$$

  The communication-to-computation ratio is therefore $\frac{1}{\beta} + \frac{2}{\mu}$, which for large matrices is asymptotically close to $\frac{1}{\beta} + \sqrt{\frac{32}{8C_D}}$. This is far from the lower bound $\sqrt{\frac{27}{8C_D}}$ derived earlier. To optimize this CCR, we could try to increase the value of β. However, increasing the parameter β implies a lower value of α, resulting in more shared-cache misses.

Algorithm 3: TRADEOFFMMRA

$offset_i = (\text{My\_Core\_Num}() - 1) \pmod{\sqrt{p}}$
$offset_j = \lfloor \frac{\text{My\_Core\_Num}()-1}{\sqrt{p}} \rfloor$
for Step $= 1$ *to* $\frac{m \times n}{\alpha^2}$ **do**
  Load a new block $C[i, \ldots, i+\alpha;\ j, \ldots, j+\alpha]$ of C in the shared cache
  for Substep $= 1$ *to* $\frac{z}{\beta}$ **do**
    $k = 1 + (Substep - 1) \times \beta$
    Load a new block row $B[k, \ldots,\ k+\beta;\ j, \ldots, j+\alpha]$ of B in the shared cache
    Load a new block column $A[i, \ldots, i+\alpha;\ 1+(k-1)\times\beta, \ldots,\ 1+k\times\beta]$ of A in the shared cache
    foreach *core* $c = 1 \ldots p$ *in parallel* **do**
      for $subi = 1$ *to* $subi = \frac{\alpha}{\sqrt{p}\mu}$ **do**
        for $subj = 1$ *to* $subj = \frac{\alpha}{\sqrt{p}\mu}$ **do**
          Load
          $C_{\mu c} = C[i + offset_i \times \frac{\alpha}{\sqrt{p}} + (subi - 1) \times \mu, \ldots, i + offset_i \times \frac{\alpha}{\sqrt{p}} + (subi) \times \mu;\ j + offset_j \times \frac{\alpha}{\sqrt{p}} + (subj - 1) \times \mu, \ldots, j + offset_j) \times \frac{\alpha}{\sqrt{p}} + (subj) \times \mu]$ in the distributed cache of core c
          for $k' = k$ *to* $k' = k + \beta$ **do**
            Load $B_c =$
            $B[k';\ j + offset_j \times \frac{\alpha}{\sqrt{p}} + (subj - 1) \times \mu, \ldots, j + offset_j \times \frac{\alpha}{\sqrt{p}} + (subj) \times \mu]$
            in the distributed cache of core c
            for $i' = i + offset_i \times \frac{\alpha}{\sqrt{p}} + (subi - 1) \times \mu$ *to*
            $i + offset_i \times \frac{\alpha}{\sqrt{p}} + (subi) \times \mu$ **do**
              Load the element $a = A[i',\ k']$ in the distributed cache of core c
              Compute the new contribution: $C_c \leftarrow C_c + a \times B_c$
          Update block $C_{\mu c}$ in the shared cache
  Write back the block of C to the main memory

Remark In the special case $\alpha = \sqrt{p}\mu$, we only need to load each $\mu \times \mu$ sub-block of C once, since a core is only in charge of one sub-block of C, therefore the number of distributed-caches misses becomes:

$$M_D = \frac{1}{p} \times \frac{mn}{\alpha^2} \times \left[\frac{\alpha^2}{\mu^2} \times \left(\mu^2 \times 1 + \frac{z}{\beta} \times 2\beta\mu \right) \right] = \frac{mn}{p} + \frac{2mnz}{p\mu}$$

In this case, we come back to the distributed-cache optimized case, and the distributed CCR is close to the bound.

- *Data access time*

  With this algorithm, we get an overall data access time of:

$$T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D} = \frac{mn + \frac{2mnz}{\alpha}}{\sigma_S} + \frac{\frac{mnz}{p\beta} + \frac{2mnz}{p\mu}}{\sigma_D}$$

  Together with the constraint $2\alpha \times \beta + \alpha^2 \leq C_D$, this expression allows us to compute the best value for parameters α and β, depending on the ratio σ_S/σ_D. Since we work under the assumption of large matrices, the first term in mn can be neglected in front of the other terms: in essence the problem reduces to minimizing the following expression:

$$\frac{2}{\sigma_S \alpha} + \frac{1}{p\sigma_D \beta} + \frac{2}{p\sigma_D \mu}$$

  The constraint $2\beta\alpha + \alpha^2 \leq C_S$ enables us to express β as a function of α and C_S. As a matter of a fact, we have:

$$\beta \leq \frac{C_S - \alpha^2}{2\alpha}$$

  Hence, the objective function becomes:

$$F(\alpha) = \frac{2}{\sigma_S \alpha} + \frac{2\alpha}{p\sigma_D (C_S - \alpha^2)}$$

  Note that we have removed the term $\frac{2}{p\sigma_D \mu}$ because it only depends on μ and therefore is minimal when $\mu = \lfloor \sqrt{C_S - 3/4} - 1/2 \rfloor$, i.e. its largest possible value.
  The derivative $F'(\alpha)$ is:

$$F'(\alpha) = \frac{2(C_S + \alpha^2)}{p\sigma_D (C_S - \alpha^2)^2} - \frac{2}{\sigma_S \alpha^2}$$

  And therefore, the root is

$$\alpha_{\text{num}} = \sqrt{C_S \frac{1 + 2\frac{p\sigma_D}{\sigma_S} - \sqrt{1 + 8\frac{p\sigma_D}{\sigma_S}}}{2\left(\frac{p\sigma_D}{\sigma_S} - 1\right)}}$$

  Altogether, the best parameters values in order to minimize the total data access time in the case of square blocks are:

$$\begin{cases} \alpha = \min\left(\alpha_{\max}, \max\left(\sqrt{p}\mu, \alpha_{\text{num}}\right)\right) \\ \beta = \max\left(\left\lfloor \frac{C_S - \alpha^2}{2\alpha} \right\rfloor, 1\right) \end{cases}$$

  where:

$$\alpha_{\max} = \sqrt{C_S + 1} - 1$$

  Parameter α depends on the values of bandwidths σ_S and σ_D. In both extreme cases, it will take a particular value indicating that tradeoff algorithm will follow the sketch of either shared-cache optimized version or distributed-caches one:

– When bandwidth σ_D is significantly higher than σ_S, the parameter α becomes:

$$\alpha_{\text{num}} \approx \sqrt{C_S} \quad \Longrightarrow \quad \alpha = \alpha_{\max}, \beta = 1$$

which means that the tradeoff algorithm chooses the shared-cache optimized version whenever distributed caches are significantly faster than the shared cache.

– On the contrary, when the bandwidth σ_S is significantly higher than σ_D, α becomes:

$$\alpha_{\text{num}} \approx \sqrt{C_S \frac{2p\sigma_D}{-\sigma_S}} \quad \Longrightarrow \quad \alpha = \sqrt{p}\mu, \beta = 1$$

which means that the tradeoff algorithm chooses the distributed optimized version whenever distributed caches are significantly slower than the shared cache (although this situation does not seem realistic in practice).

4 Performance evaluation on CPU

We have designed and analytically evaluated three algorithms minimizing shared cache misses, distributed cache misses, and the overall time spent in data movement. In this section, we provide an implementation of these algorithms on a quad-core CPU platform. Section 5 is the counterpart for a GPU platform.

The goal of this experimental section is to evaluate the impact of real hardware caches, which are not fully associative in practice. Also, we made the hypothesis that the cost of cache misses would dominate execution time, and this hypothesis need be confronted to reality.

4.1 Experimental setting

Hardware platform The hardware platform used in this study is based on two Intel Xeon E5520 processors running at 2.26 GHz. The Intel Xeon E5520 processor is a quad core processor with a 3 level cache hierarchy, L1 and L2 caches being private while L3 cache is shared across all cores. Quite naturally, we ignore the L1 cache and focus on the L2 cache that plays the role of the distributed cache C_D, while the L3 cache will be denoted C_S.

However, despite this simplification, there remains differences between this architecture and the theoretical model. First, these caches are not fully associative : C_D is 8-way associative while C_S is 16-way associative. Moreover, they do not implement an optimal omniscient data replacement policy, but instead they operate with the classical LRU policy. Furthermore, C_D is used to cache both data and instructions, while we considered only data in our study.

The system also embeds 16GB of memory, 12GB being connected to the first processor, and 4GB to the second. This somewhat peculiar memory repartition is explained by the fact that our experiments will be conducted on the first processor only, thus giving it more local memory seems natural. Detailed characteristics of the system are displayed in Table 1.

Software framework For the experiments, the following software stack has been used:

- GCC 4.4.2 for compilation
- *Numactl* and *Libnuma* [7] for affinity
- Hwloc [8] for hardware topology detection
- GotoBLAS2 [9] for BLAS Level 3 routines
- Intel MKL [10] for BLAS Level 3 routines
- Intel VTune [11] for cache misses sampling

Processors	2x Intel Xeon E5520
Core arrangement	Processor 0: core 0-3 Processor 1: core 4-7
Core frequency	2.26GHz
QPI bandwidth	23.44 GB/s per link
L1 cache size	32 KB (data) / 32 KB (instruction)
L2 cache size	256 KB
L3 cache size	8 MB
Cache line size	64 B
Page size	4 KB / 2 MB (small/huge pages)
L1 data TLB	48/32 entries for small/huge pages
L2 TLB	512 entries for small pages
Memory type	8x 2 GB DDR3-1066, registered, ECC 3 channels (12 GB)on processor 0 2 channels (4 GB) on processor 1 8.53 GB/s per channel
Operating System	Ubuntu 8.04, Kernel 2.6.28.10 NUMA

Table 1: System configuration

Algorithms In the following experiments, we compare a large number of algorithmic versions. In addition to the vendor library, we have implemented five algorithms: two reference algorithms, the ScaLaPack outer-product [2] and the equal-layout algorithm [3], and the three optimized algorithms SharedMMRA, DistributedMMRA and TradeoffMMRA. Each of the latter three algorithms comes in two versions that differ in the size of the cache allocation: either we use the whole cache, as in the ideal versions described in Section 3, or we use only half of it, the other half being used as a buffer for LRU policy. The equal-layout algorithm is declined along four versions: shared/distributed version, and ideal/LRU cache. Altogether, this leads to 12 versions, which are labeled as follows:

- Parallel is the vendor library (MKL or GotoBLAS2) parallel matrix product implementation
- OuterProduct is the outer product Algorithm
- SharedOpt is SharedMMRA using the entire C_S
- SharedOpt-LRU is SharedMMRA using half of C_S, the other half being used as a buffer for LRU policy
- DistributedOpt is DistributedMMRA using the entire C_D
- DistributedOpt-LRU is DistributedMMRA using half of C_D (other half for LRU)
- Tradeoff is TradeoffMMRA using the entire C_S and C_D
- Tradeoff-LRU is the TradeoffMMRA using half of C_S and C_D (other half for LRU)
- DistributedEqual is the equal-layout algorithm [3] using the entire C_D
- DistributedEqual-LRU is the equal-layout algorithm [3] using half of C_D (other half for LRU)
- SharedEqual is the equal-layout algorithm [3] using the entire C_S
- SharedEqual-LRU is the equal-layout algorithm [3] using half of C_S (other half for LRU)

4.2 Performance results: execution time

The first performance metric that we naturally evaluate is the running time. Since our analysis is based under the assumption that execution time is limited by the cost of caches misses, we aim here at validating this hypothesis.

In Figures 4(a) and 4(b), we plot the runtime of every algorithms using square blocks of 96 by 96 elements for BLAS calls. Indeed, it is widely assumed that memory latencies are hidden when using that block size. We also plot the runtimes of libraries when called on the whole matrices.

With both libraries, every algorithm offers almost the same performance: the gap between the slowest algorithm and the fastest one is only 5%. Moreover, using those parameters, all algorithms are able to reach up to 89% of the performance of libraries, TRADEOFFMMRA-LRU being the best performing algorithm in that experiment.

However, this low difference between each algorithms is increased when calling BLAS on smaller blocks. On Figures 4(c) and 4(d), we did use 48 by 48 blocks of matrix coefficients as the block unit of our algorithms. We can see that the algorithms behave differently to that change, and that algorithms taking C_S into consideration offer the best performance. TRADEOFF-LRU being the best of them.

Altogether, the performance offered by our algorithms do not hold the comparison with both vendor libraries. Considering the fact that libraries are really low-level implementations whereas we aimed at design higher-level strategies, it seems natural. However, developing such low-level libraries requires a huge effort for the programmers, and everything needs to be done from scratch for each architecture. In addition, the scalability of both our algorithms and of these libraries should also be checked, but it is still hard to find general purpose x86 multicore processors with more than 8 cores, thus postponing the assessment for further work.

Furthermore, if we restrict the comparison to the algorithms that we have implemented, there is a significant difference between them only for the smallest block sizes, which are not used in practice because with such blocks, data reads and writes are not overlapped with computations.

In conclusion, cache misses do not seem to have an important impact on run-times (but M_S appears more important than M_D, as expected). Several other mechanisms interact in modern multi-core processors, and a cache miss do not necessarily leads to a longer run-time than a cache hit because it might improve prefetching. Unfortunately, refining the model and taking other metrics like TLB misses of prefetching into consideration would render the analysis intractable.

Finally, we observe that three blocks of 96 by 96 matrix elements entirely fill C_D, thus diminishing the ability of MMRA to handle distributed caches properly, because the value of μ is obviously forced to 1. For the same reason, the value of λ is also limited. From this observation we think that MMRA would benefit a lot from larger caches, increasing the decision space.

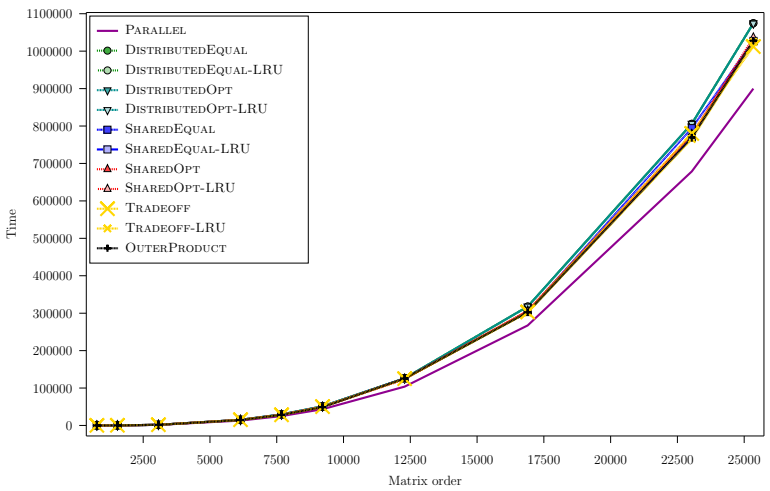
4.3 Performance results: cache misses

We also study the behavior of our algorithms in terms of cache misses. The objective is twofold: we aim at validating the behavior of our algorithms, while giving a more precise analysis of the impact of cache misses on run-time.

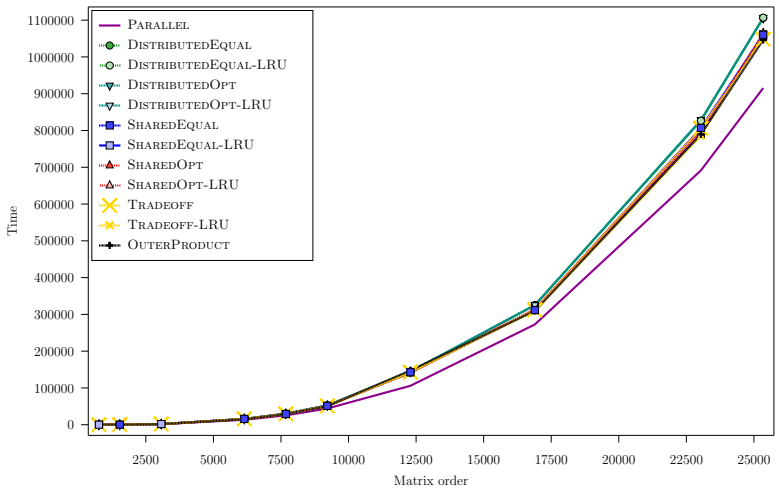
Figures 5(a), 5(b), 5(c) and 5(d) show that for M_S, we observe what we expected: algorithms focusing on C_S are the best alternatives and algorithms focusing on distributed caches are the worst alternatives. For big matrices, MMRA sometimes performs better than libraries on this precise metric. Moreover, diminishing block size increases the gap between algorithms, similarly to what we observe for run-time.

Interestingly, in some cases, increasing the size of the matrices decreases the number of shared cache misses. This behavior is due to the fact that C_S is 16-way associative whereas we assume full-associativity in our study. Fully-associative caches are extremely costly, and makes the mapping between cache and memory a combinatorial problem. That is why real hardware rather implement limited associativity caches. In those cases, the associativity has a non negligible impact on M_S.

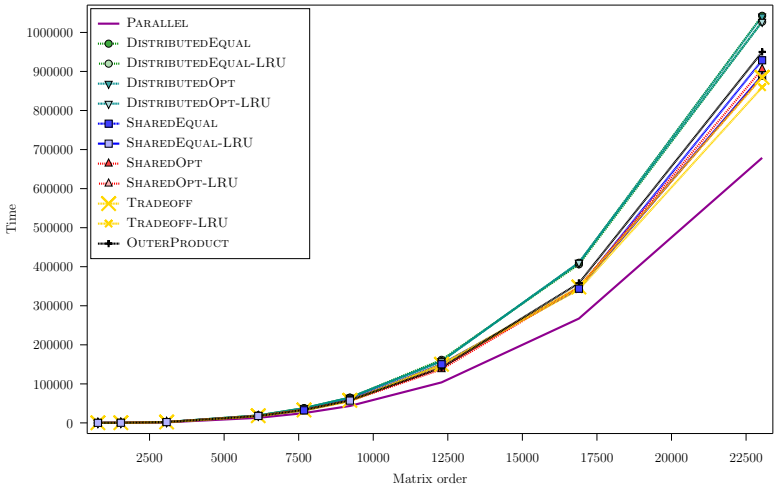
However, even though we are able to reduce M_S, the run-time of our algorithms does not follow the same trend. On Figures 6(a), 6(b), 6(c) and 6(d), we measure the number of shared caches



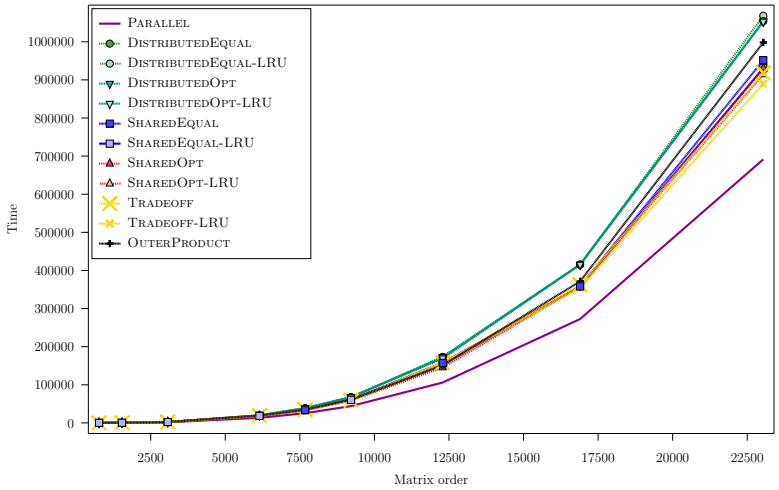
(a) $q = 96$, runtimes using GotoBLAS2 library.



(b) $q = 96$, runtimes using MKL library.

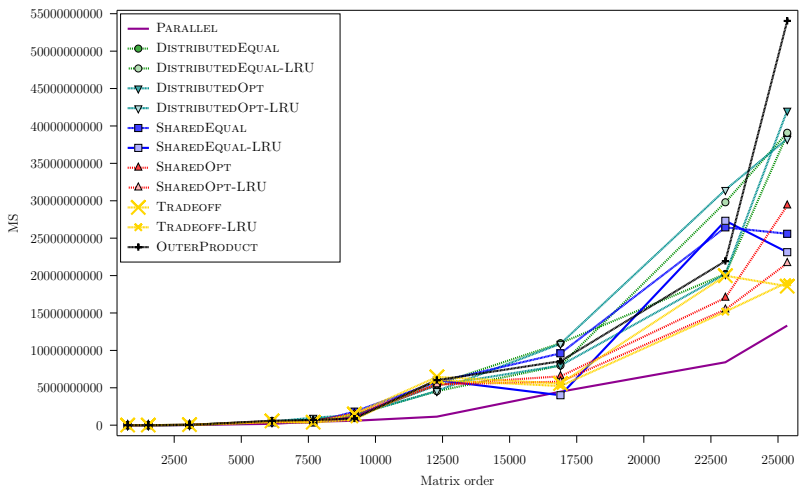


(c) $q = 48$, runtimes using GotoBLAS2 library.

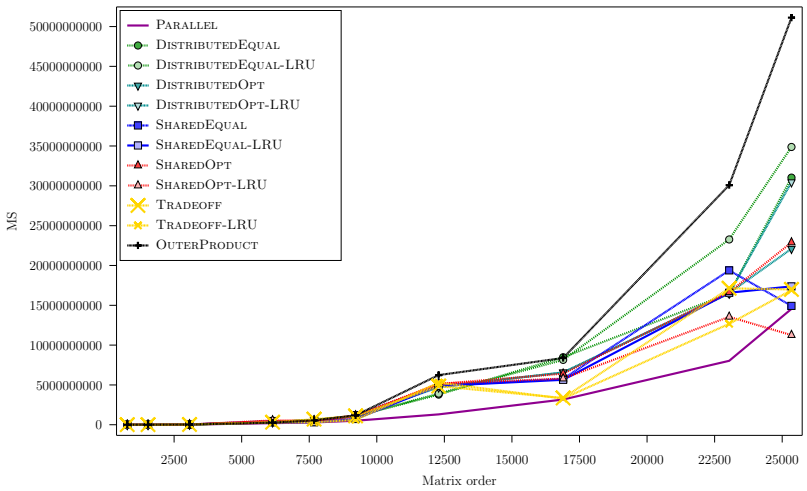


(d) $q = 48$, runtimes using MKL library.

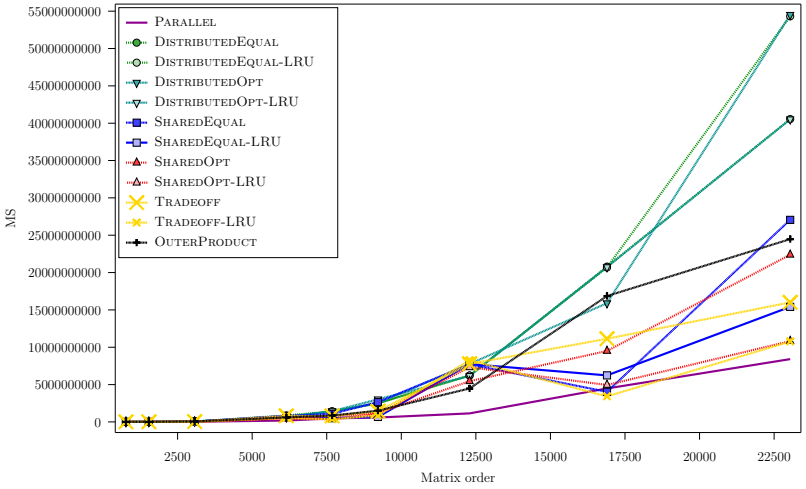
Figure 4: Runtimes according to matrix order.



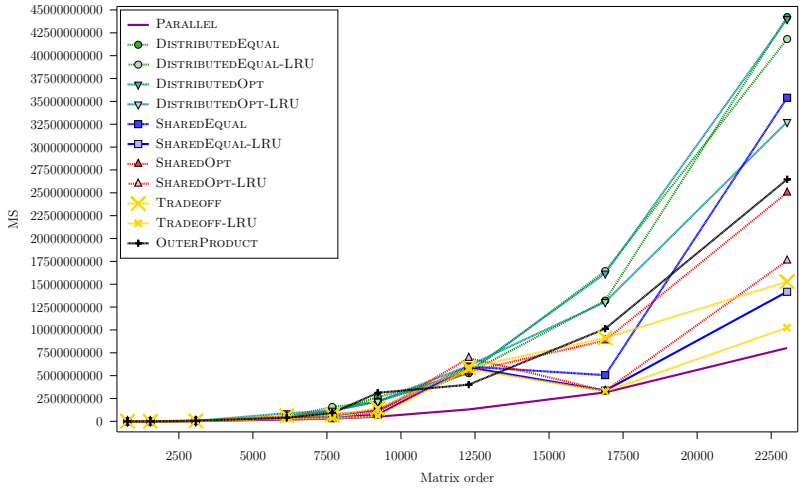
(a) $q = 96$, total M_S using GotoBLAS2 library.



(b) $q = 96$, total M_S using MKL library.

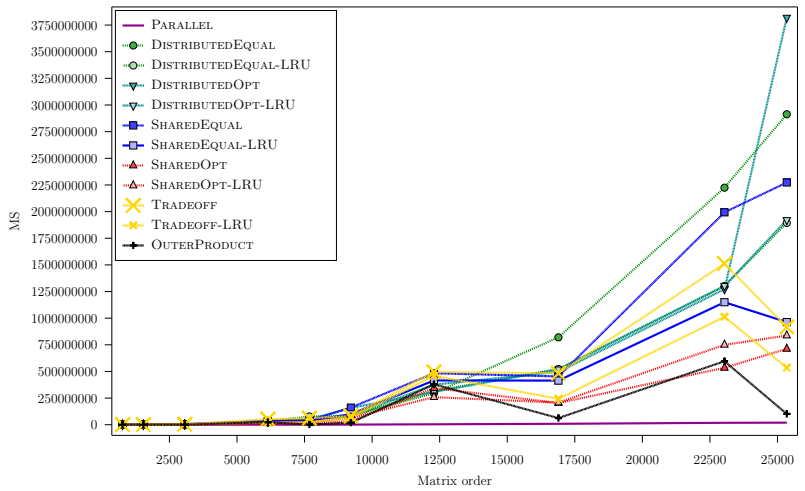


(c) $q = 48$, total M_S using GotoBLAS2 library.

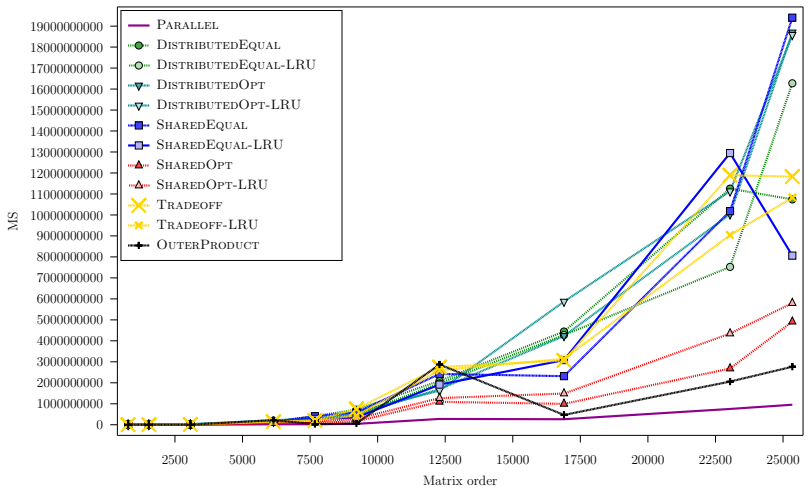


(d) $q = 48$, total M_S using MKL library.

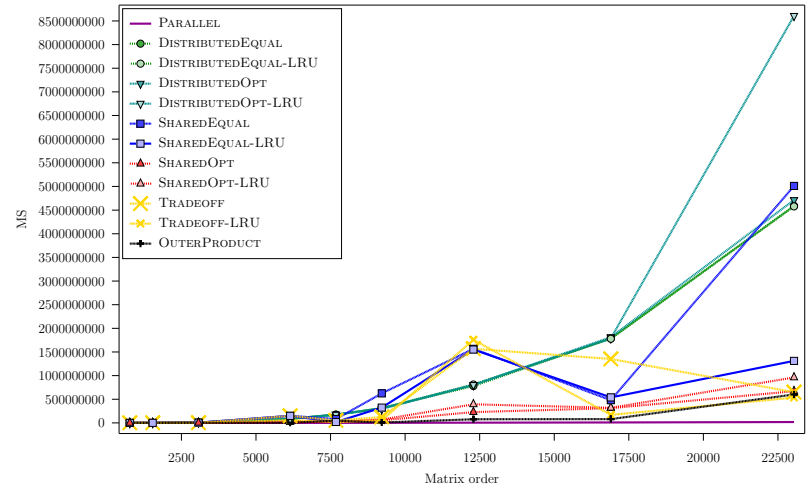
Figure 5: Total shared cache misses M_S according to matrix order.



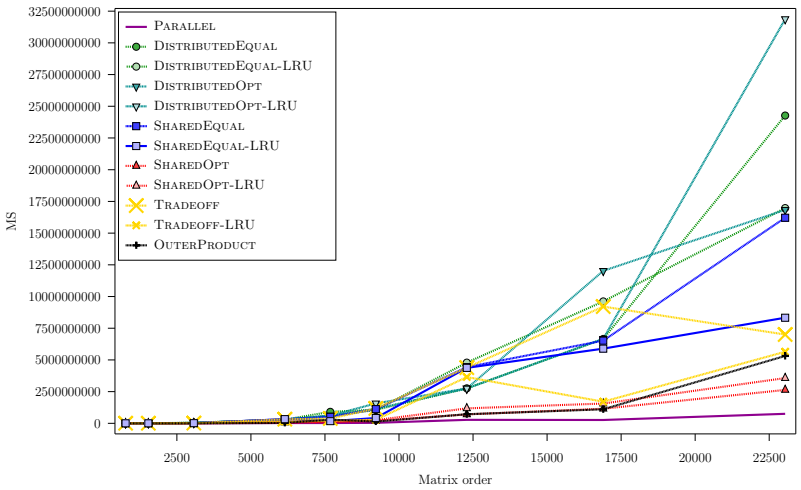
(a) $q = 96$, non prefetched M_S using GotoBLAS2 library.



(b) $q = 96$, non prefetched M_S using MKL library.



(c) $q = 48$, non prefetched M_S using GotoBLAS2 library.



(d) $q = 48$, non prefetched M_S using MKL library.

Figure 6: Non prefetched shared cache misses M_S according to matrix order.

misses that are not prefetched by the processor. Theses misses are the ones that actually slowdown the processor, thus having a direct impact on run-time. Our algorithms suffer from their more complicated access pattern, which drastically reduce prefetching. Conversely, libraries and outer product keep their access patterns simple enough, to fully benefit from hardware prefetchers, thus making most of their cache misses for free.

When considering the total number of distributed caches misses, the experiments (see Figures 7(a), 7(b), 7(c) and 7(d)) show that with blocks of 96 by 96 elements, the difference in performance between each algorithm is low, and algorithms focusing on C_D do not significantly reduce the number distributed cache misses. Surprisingly in that case, both libraries make significantly higher M_D.

When we use smaller blocks, the difference in performance increases but we cannot observe the dominance of DistributedMMRA, though it is theoretically optimal. This small impact of algorithms on M_D is probably due to the fact that C_D are not dedicated to data only but also cache instructions. Therefore we do not benefit of the entire space for our tiling algorithms. It also seems that the behavior of data replacement policy is biased because data gets evicted in order to load instructions.

We now focus on the number of unprefetched distributed caches misses depicted in Figures 8(a), 8(b), 8(c) and 8(d). Libraries clearly prefetch most part of their distributed cache misses and therefore pay significantly less misses than the other algorithms. Moreover, the algorithms focusing on M_D are the best alternatives with both block sizes, though results using smaller blocks are more regular.

Furthermore, for distributed cache oriented algorithms, we may face false sharing issues. This problem occurs when two matrix elements of two different blocks of C are within the same cache line. The line gets modified by a core, thus triggering coherency protocol between the cores sharing that same line of cache. On Intel Nehalem processors, the cost is twice that of an unshared access [12], thus implying significant impact on run-time. Note that here again, the lower associativity of C_D has a higher impact on cache misses and significantly increases M_D.

Finally, it is difficult to ensure total ownership on the processor to a given application. Therefore, a few other processes sometimes "trash" the cache with their data, thus evicting matrix data from cache. On top of this, BLAS libraries often reorganize data so as to fulfill their own requirements, by sometimes making copies. All in all, these two trends disturb the behavior of our algorithms and increase the number of cache misses at each level.

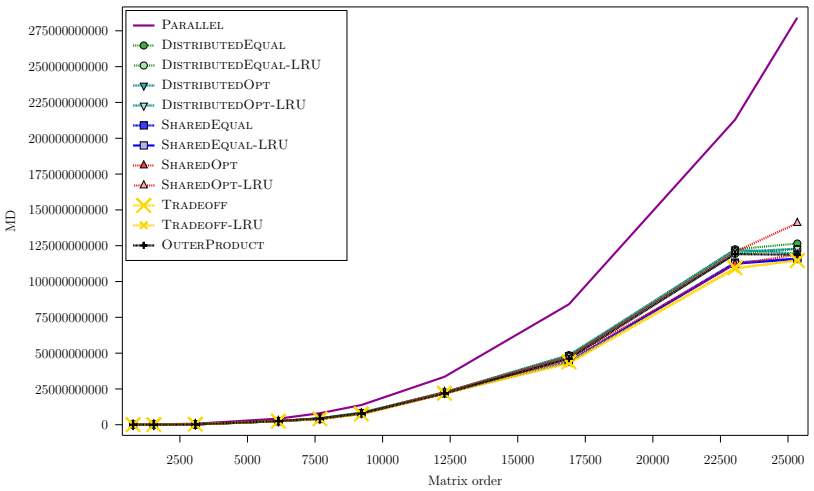
As a conclusion, these experiments show that on current CPU architectures, it is hard to precisely predict and thus control the cache behavior of a given algorithm. Well known algorithms like matrix product, where memory behavior could be precisely handled, could benefit from having total control over caches. Having such a control over caches is equivalent to consider them as fast local memories. Such memories are often encountered on today's accelerator architectures like the Cell processor [13] or GPUs [14] [15]. We study the latter platform in Section 5.

5 Performance evaluation on GPU

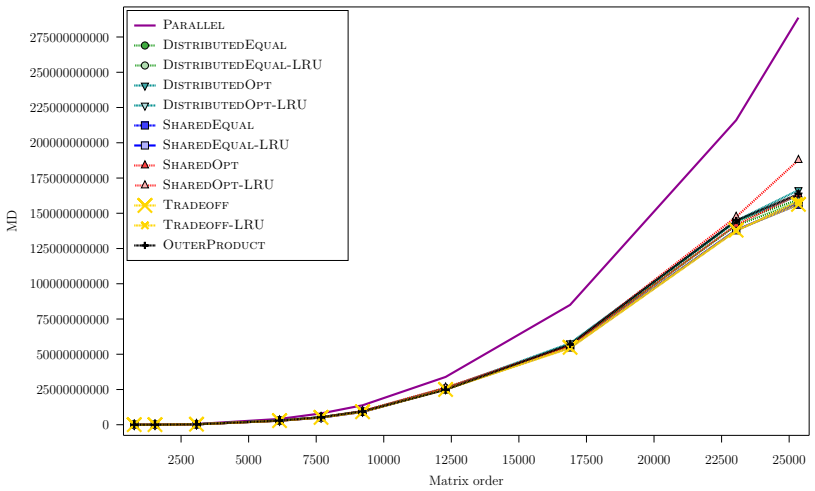
As outlined above, one of the main drawbacks of current CPU architectures is the fact that caches are always managed by hardware, and hence cannot be used are regular low-latency memories. However, in the context of simple linear algebra kernels like matrix product, data could be handled manually at reasonable cost. Such low-latency memories can be found and programmed on GPUs. In this part, we hence focus on the adaptation of MMRA to GPUs and assess its performance on such hardware platforms.

5.1 Experimental setting

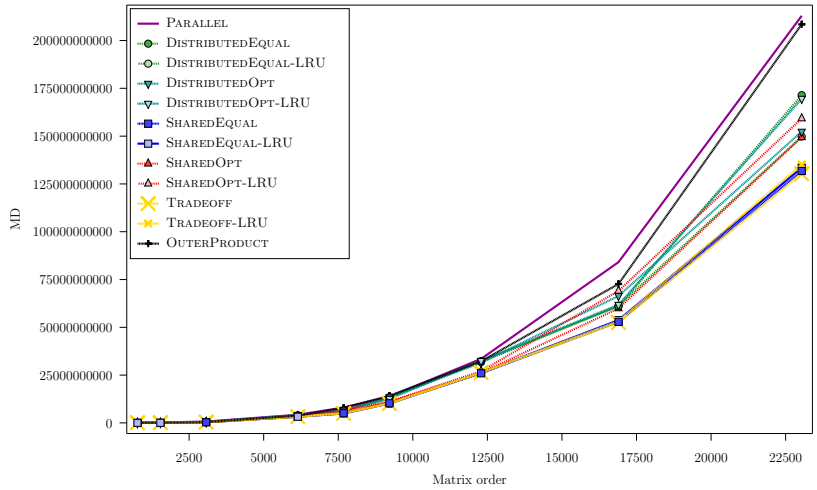
For the experiments, we use the same hardware platform as in Section 4.1, with an additional GPU card based on GT200 architecture from NVidia. The system runs on Yellow Dog Enterprise Linux for NVIDIA CUDA based on the 2.6.18 Linux kernel. The GT200 architecture is widespread



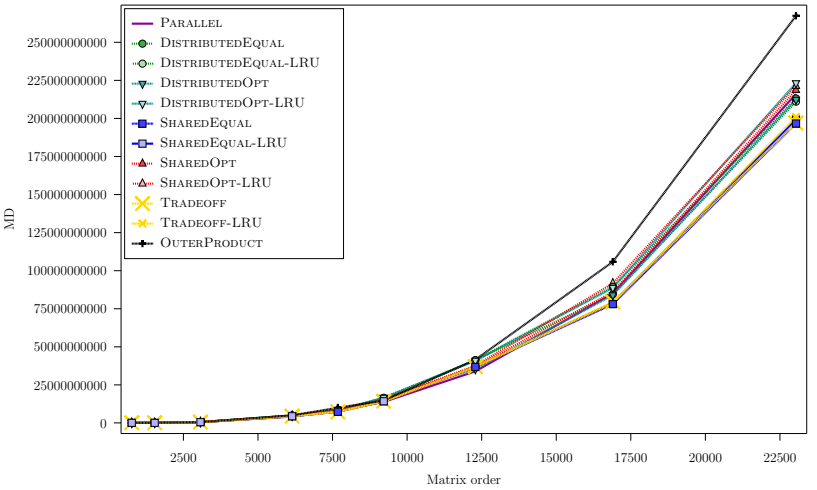
(a) $q = 96$, total M_D using GotoBLAS2 library.



(b) $q = 96$, total M_D using MKL library.

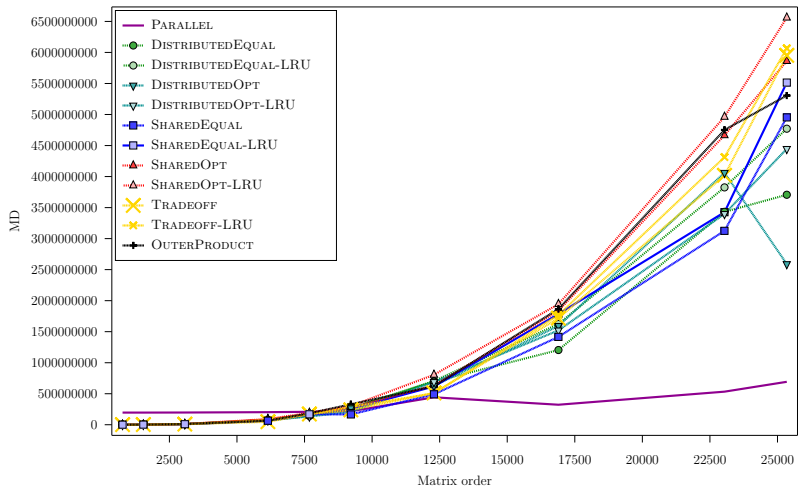


(c) $q = 48$, total M_D using GotoBLAS2 library.

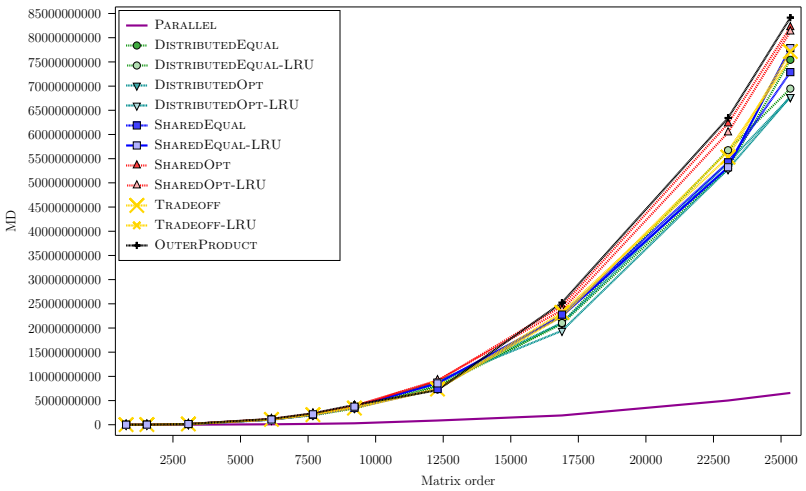


(d) $q = 48$, total M_D using MKL library.

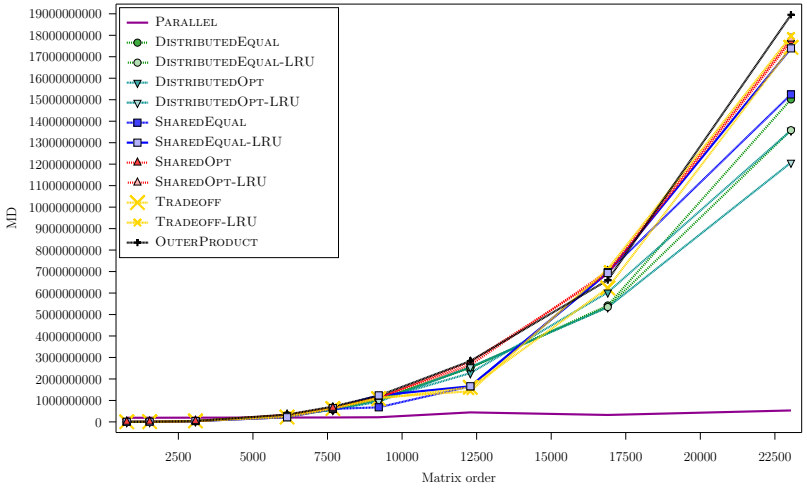
Figure 7: Total distributed cache misses M_D according to matrix order.



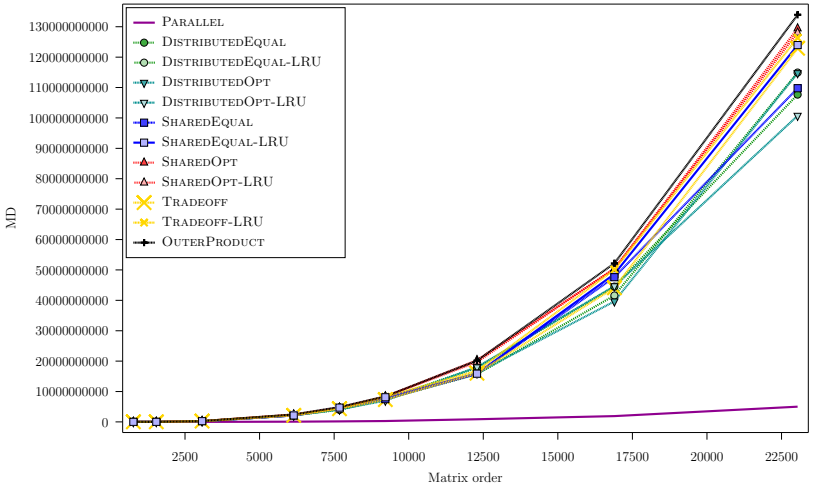
(a) $q = 96$, non prefetched M_D using GotoBLAS2 library.



(b) $q = 96$, non prefetched M_D using MKL library.



(c) $q = 48$, non prefetched M_D using GotoBLAS2 library.



(d) $q = 48$, non prefetched M_D using MKL library.

Figure 8: Non prefetched distributed cache misses M_D according to matrix order.

in the HPC community through Tesla GPU computing devices, and is therefore a representative candidate for this study.

The card is a GeForce GTX 285, which embeds 240 cores running at 1.48 GHz, and 2GB of GDDR3 memory; CUDA 2.3 has been chosen as the programming environment. The 240 cores or Stream Processors (SPs) are grouped in 30 clusters called multi-processors (MPs). These multi-processors have a "shared" memory of 16KB shared across every SPs within the multi-processor. They also have 16384 32-bit private registers distributed among the SPs. Finally, the card has a memory space called "global" memory (of 2GB with in this setup) which is shared among every SPs, but which is significantly slower than "shared" memory and registers. The interested reader can find more details on GT200 and CUDA thread management model in [16].

5.2 Adaptation of MMRA to the GT200 Architecture

The hardware architecture of GPUs leads to revisit our tiling algorithm in order to cope with its specificities. GPUs have several levels of memory, including on-board RAM memory as well as on-chip memory, which is order of magnitudes faster than on-board memory. This speed heterogeneity should therefore be taken into account, by fine tuning memory accesses at every level, thus leading us to choose TradeoffMMRA. The main idea of this adaptation is to consider the on-board RAM memory (or "global memory") as a shared cache of size C_S whereas each multi-processor will have a distributed cache of size C_D located in on-chip memory (which are called "shared" memories and registers in CUDA vocabulary).

Moreover our algorithm was originally designed for p^2 processors, which is not flexible and leads to underusage of the hardware. Therefore, we decided to modify TradeoffMMRA so as to handle $p \times q$ processors. Instead of tiling the $\alpha \times \alpha$ block of C in p^2 subblocks of size $\alpha/p \times \alpha/p$, we rather cut the block of C in $p \times q$ subblocks of size $\alpha/p \times \alpha/q$ (represented as light gray blocks in Figure 9).

Furthermore, we need to reserve additional memory in C_S for blocks of A and B in order to overlap PCI-E transfers between host and GPU with computation. We thus need to consider this in the computation of α. The new equation becomes:

$$\alpha^2 + 4\alpha\beta \leq M$$

For the computation of μ, it is necessary to consider both the fact that GPUs are efficient only with a high number of concurrent threads, and the fact that the block of C is not partitioned in square tiles anymore.

On GT200, the best hardware occupation is obtained when running more than 512 threads per multi-processor. Note that on GPUs, threads truly are lightweight, which means that in the context of this study, a thread will process only a few matrix elements. Let γ be the number of threads around the first dimension (i.e. number of lines), and τ be the number of threads around the second dimension.

In order to handle rectangular tiles, let η be the number of columns of the block in C_D, and χ be its counterpart for the number of rows. Altogether, η must divide α/q and be a multiple of τ; while χ must divide α/p and be a multiple of γ. Such blocks are depicted in dark grey in Figure 9. All in all, each thread will process $(\eta \times \chi)/(\tau \times \gamma)$ matrix elements of C.

However, contrarily to the original TradeoffMMRA, $\chi \times \eta$ subblocks of C will not reside in the same memory than subblocks of A and B: in fact, C_D is no longer a homogeneous memory space. This feature comes from the fact that C is not shared among cores (or SPs), and can therefore be stored in the private registers available on GT200. The red squares in Figure 9 are the elements of C loaded in registers and processed by a given thread T.

Conversely, A and B coefficients are shared, hence stored in "shared" memory. Moreover, for efficiency reasons, instead of loading only one column of A and one row of B, block-columns and block-rows of respectively $\gamma \times \chi$ and $\eta \times \gamma$ elements are loaded. Each thread loads therefore χ/γ elements of A and η/τ elements of B in "shared" memory at each step. For a given thread T, it

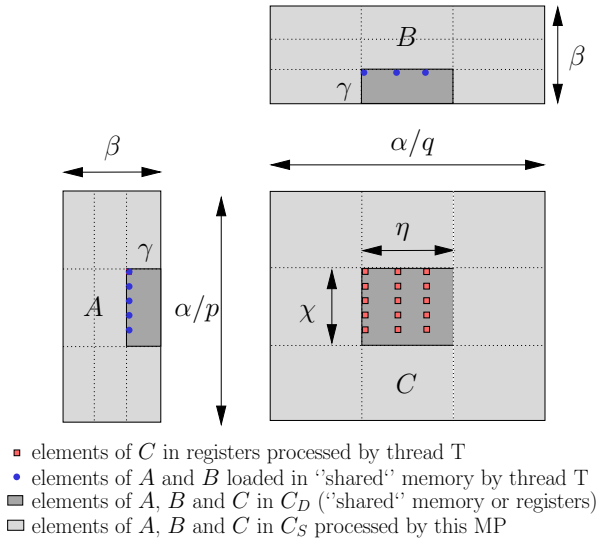


Figure 9: Tiling of matrices within a multi-processor

Multi-processors on lines	$p = 6$
Multi-processors on columns	$q = 5$
Thread count on lines	$\gamma = 16$
Thread count on columns	$\tau = 32$
Total number of threads	$\tau \times \gamma = 512$
Number of lines per MP	$\chi = 80$
Number of columns per MP	$\eta = 96$
Registers used by elements of C per MP	50%
"Shared" memory used by elements of A and B per MP	70%

Table 2: Parameters for GeForce GTX 285 (GT200)

corresponds to the blue circles depicted in Figure 9. Altogether, the parameter used by MMRA on GTX285 are given in Table 2.

5.3 Performance results: execution time

In the following experiments, the results are obtained using three different algorithms:

- TRADEOFF is the adaptation of MMRA to GT200 architecture.
- SHAREDEQUAL allocates one third of C_S to each matrix. This setup is built on top of the MMRA kernel with different tiling parameters.
- CUBLAS calls the so-called vendor library made by NVidia. The calls are made in the same order than the classical outer product algorithm.

Every result given in the following corresponds to the average of 10 experimental results.

The first experiment focuses on the efficiency as well as the scalability of TRADEOFF compared to NVidia's library CUBLAS. As depicted on Figure 10, the run times of each algorithm is depicted according to the matrix dimension. TRADEOFF is the fastest algorithm in two cases, reducing the time required by CUBLAS by up to 13%. In the other cases, TRADEOFF is slower than CUBLAS by up to 40%. SHAREDEQUAL always is slower to TRADEOFF, but remains close.

Note that the runtime of CUBLAS does not scale linearly when the size of the matrices increases. In order to understand this behavior, the execution of each case was profiled using CUDA

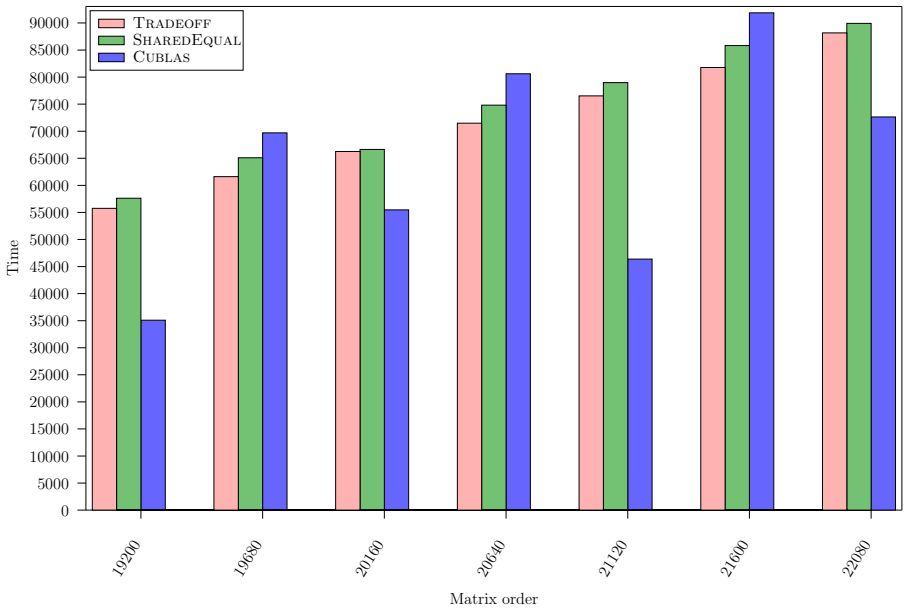


Figure 10: Runtimes on GTX285

19200	*sgemmNN*
19680	*sgemm_main_gld_hw_na_nb*
20160	*sgemm_main_gld_hw_na_nb_fulltile*
20640	*sgemm_main_gld_hw_na_nb*
21120	*sgemmNN*
21600	*sgemm_main_gld_hw_na_nb*
22080	*sgemm_main_gld_hw_na_nb_fulltile*

Table 3: CUBLAS kernel used in function of matrix size

Visual Profiler, allowing the identification of the actual computation kernel call underlying below CUBLAS calls. As expected, different kernels are called, as shown on Table 3, and every case where Tradeoff outperforms Cublas, Cublas uses the same CUBLAS computation kernel, which does not make any usage of GPU's specific hardware features like texture units. Moreover, in the adaptation of TradeoffMMRA to GPU, this kind of hardware units were ignored on purpose, in order to stay close to the theoretical model, and to keep the independence of the approach with respect to to the underlying hardware architecture. However, in order to be able to compete with vendor's libraries like CUBLAS, it would be necessary to use these specific dedicated hardware units (but it is not in the scope of this study).

5.4 Performance results: cache misses

In the next experiment, the number of shared cache misses M_S committed by each algorithm is evaluated. Results are depicted on Figure 11. As expected, Tradeoff provide the best result since it makes up to 70% less shared cache misses than Cublas, thus proving the efficiency of our approach on GPUs. SharedEqual provides the same results than Cublas since it uses the same tiling at this level of cache.

On Figure 12, the actual time spent by the GPU moving data in order to serve the shared cache (i.e. the time penalty associated to M_S) is depicted. Unsurprisingly, Tradeoff spend the least time, reducing the time spent by Cublas and SharedEqual by up to 32%.

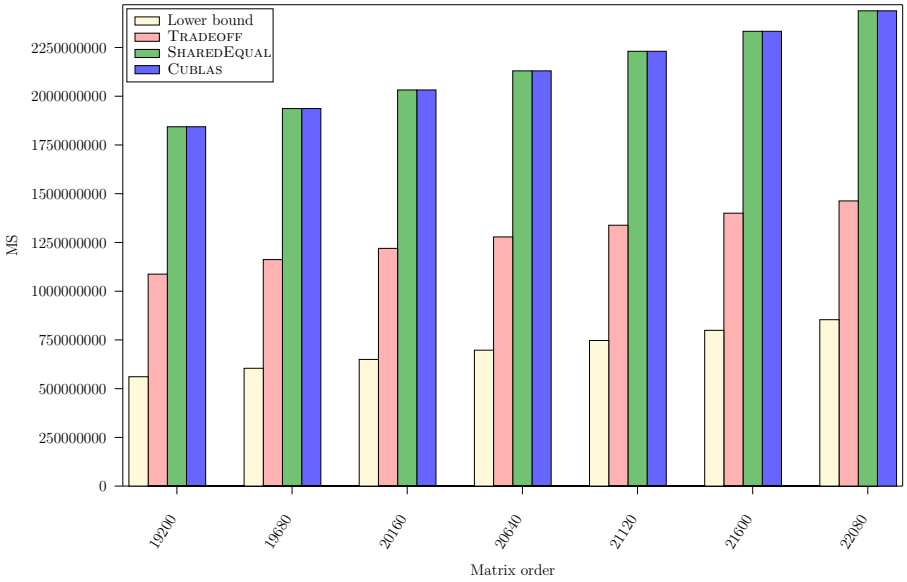


Figure 11: M_S on GTX285

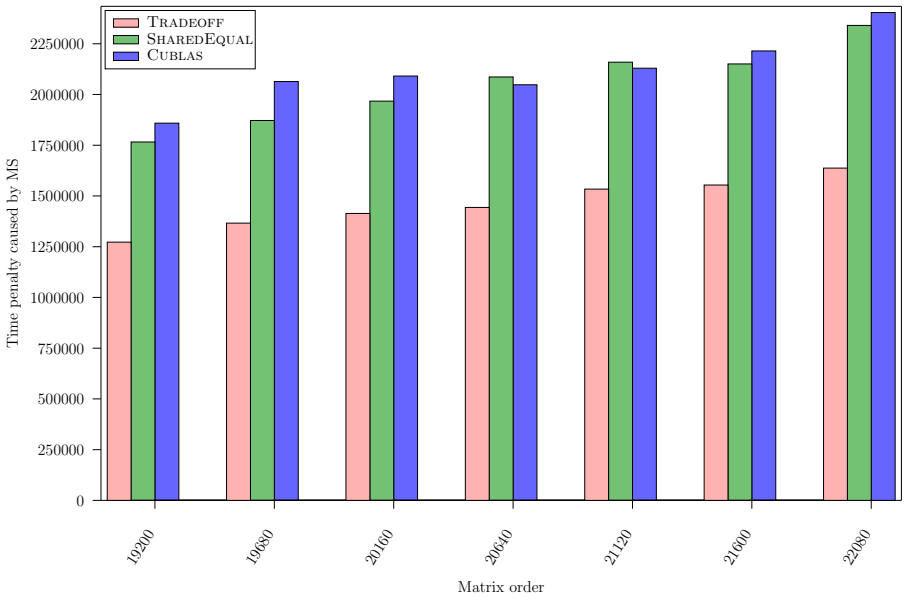


Figure 12: Time penalty caused by M_S on GTX285

Finally, in the last experiment, whose results are given in Figure 13, the number of distributed caches misses M_D committed by a multi-processor is depicted. In that case, the lowest number of M_D is given by SharedEqual. This is due to the fact that, even though they share the same kernel, SharedEqual uses a higher value of β than Tradeoff. However in most cases, Tradeoff ties SharedEqual, depending on its value of β. Cublas experiences between 1.9 and 3.4 times more distributed caches misses. However, if we consider for instance the performance on a matrix of size 21120, Cublas offers the best execution time while encountering the highest M_D. Therefore, the impact of M_D on performance seems to be low.

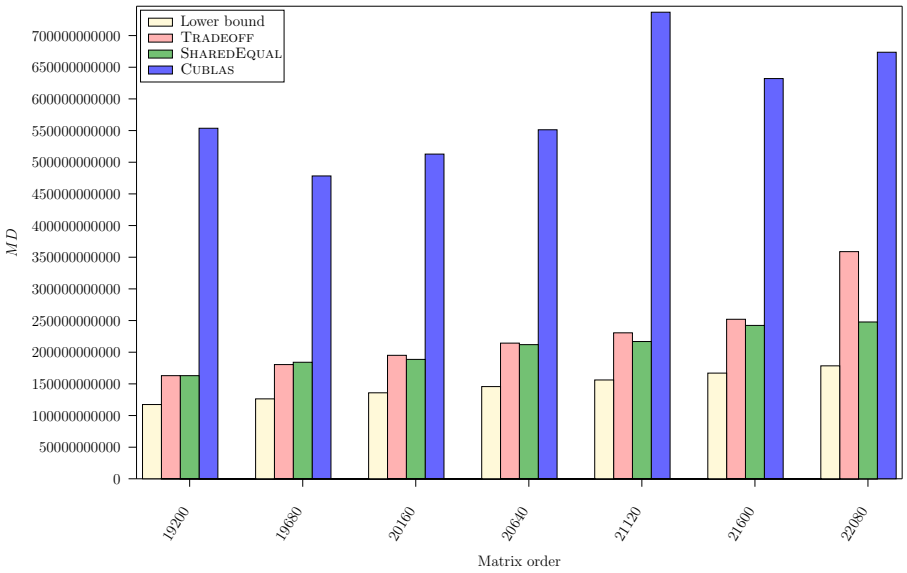


Figure 13: M_D on GTX285

Altogether, the approach used by the MMRA proves its efficiency on alternative architectures like GPUs leading to cache efficient computation kernels. As the memory wall is closing in, the need of such algorithms is getting more important. This study proves that good results can be obtained using a high level, analytical and architecture-independent approach.

6 Conclusion

In this report, we have proposed cache-aware matrix product algorithms for multicore processors. Using a simple yet realistic model for multicore memory layout, we have extended a lower bound on cache misses, and proposed cache-aware algorithms. For both types of caches, shared an distributed, our algorithms reach a CCR which is close to the corresponding 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.

We have also provided an extensive set of experimental results to compare the effective performance of the previous algorithms on actual multicore platforms. The objective was twofold: (i) assess the impact of cache misses onto runtime performance; and (ii) validate our high-level approach by comparing the algorithms to the vendor library routines. We have obtained mixed results with CPU platforms: due to intensive prefetching, cache misses have a lesser impact than the theoretical analysis has predicted. On the contrary, results with GPU platforms were quite encouraging and have nicely demonstrated the accuracy of our model, and the efficiency of our architecture-independent partitioning approach. Low-latency memories such as those provided by GPUs are much more promising devices than general purpose LRU caches for the tiling and partitioning strategies that lie at the heart of state-of-the-art linear algebra kernels.

Future work will proceed along two main directions. On the pure experimental side, we would like to run comparisons on larger scale platforms, with 32 or 64 cores. On the algorithmic side, we envision to design 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.

Acknowledgments. The authors are with Université de Lyon, France. L. Marchal is with CNRS, France. Y. Robert is with the Institut Universitaire de France.

References

[1] L. E. Cannon, A cellular computer to implement the kalman filter algorithm, Ph.D. thesis, 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, 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, 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, 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] D. Ironya, S. Toledo, A. Tiskin, Communication lower bounds for distributed-memory matrix multiplication, J. Parallel Distributed Computing 64 (9) (2004) 1017–1026.

[7] A. Kleen, A numa api for linux, http://andikleen.de/ (2005).

[8] F. Broquedis, J. Clet Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications, in: IEEE (Ed.), PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, Pisa Italie, 2010.
URL http://hal.inria.fr/inria-00429889/en/

[9] GotoBLAS2, http://www.tacc.utexas.edu/tacc-projects/gotoblas2/.

[10] Intel Math Kernel Library, http://software.intel.com/en-us/intel-mkl/ (2010).

[11] Intel VTune Performance Analyzer, http://software.intel.com/en-us/intel-vtune/ (2010).

[12] T. Rolf, Cache organization and memory management of the Intel Nehalem computer architecture, Research report, University of Utah Computer Engineering (2009).
URL http://rolfed.com/nehalem/nehalemPaper.pdf

[13] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, D. J. Shippy, Introduction to the cell multiprocessor, IBM Journal of Research and Development 49 (4-5) (2005) 589–604.

[14] ATI stream technology, Official website, AMD (2010).
URL http://www.amd.com/stream

[15] CUDA Programming Model, Official website, NVIDIA (2010).
URL http://www.nvidia.com/object/what_is_cuda_new.html

[16] CUDA C Programming Guide, Technical documentation, NVIDIA (2010).
URL http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_2.3.pdf

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399