A high-performance matrix-matrix multiplication methodology for CPU and GPU architectures

KELEFOURAS, Vasileios <http://orcid.org/0000-0001-9591-913X>, KRITIKAKOU, Angeliki, MPORAS, Iosif and VASILEIOS, Kolonias

Published version

KELEFOURAS, Vasileios, KRITIKAKOU, Angeliki, MPORAS, Iosif and VASILEIOS, Kolonias (2016). A high-performance matrix-matrix multiplication methodology for CPU and GPU architectures. Journal of Supercomputing, 72 (3), 804-844.

Noname manuscript No.
(will be inserted by the editor)

A high performance Matrix-Matrix Multiplication Methodology for CPU and GPU architectures

Vasilios Kelefouras, Angeliki Kritikakou,
Iosif Mporas, Vasilios Kolonias

the date of receipt and acceptance should be inserted later

Abstract Current compilers cannot generate code that can compete with hand-tuned code in efficiency, even for a simple kernel like Matrix-Matrix Multiplication. A key step in program optimization is the estimation of optimal values for parameters such as tile sizes and number of levels of tiling. The scheduling parameter values selection is a very difficult and time-consuming task since parameter values depend on each other; this is why they are found by using searching methods and empirical techniques. To overcome this problem, the scheduling sub-problems must be optimized together, as one problem and not separately.

In this paper a Matrix-Matrix Multiplication methodology is presented where the optimum scheduling parameters are found by decreasing the search space theoretically while the major scheduling sub-problems are addressed together as one problem and not separately according to the hardware architecture parameters and input size; for different hardware architecture parameters and/or input sizes, a different implementation is produced. This is achieved by fully exploiting the software characteristics (e.g., data reuse) and hardware architecture parameters (e.g., data caches sizes and associativities), giving high quality solutions and a smaller search space. This methodology refers to a wide range of CPU and GPU architectures.

Keywords Matrix-Matrix Multiplication, Data reuse, optimization, SIMD, memory hierarchy, loop tiling

1 Introduction

Matrix-Matrix Multiplication (MMM) is an important kernel in most varied domains and application areas. Its performance is of great practical importance and it highly depends on memory management; the most performance critical

Address(es) of author(s) should be given

sub-problems are these of finding the schedules with the minimum numbers of i) L1 data cache accesses, ii) L2 data cache accesses, iii) L3 data cache accesses, iv) main memory data accesses, v) addressing instructions. The above sub-problems depend on each other (e.g., a decrease on the number of L2 data cache accesses will consequently increase the number of L1 data cache accesses) and this is why they should be addresses together as one problem and not separately (in [1], a methodology for Matrix-Vector Multiplication is given).

Toward this, much research has been done, either to simultaneously optimize only two phases, e.g., register allocation and instruction scheduling (the problem is known to be NP-complete) [2] [3] or to apply predictive heuristics [4] [5]. Nowadays compilers and related works, apply either iterative compilation techniques [6] [7] [8] [9], or both iterative compilation and machine learning compilation techniques to restrict the configurations' search space [10] [11] [12] [13] [14] [15]. A predictive heuristic tries to determine a priori whether or not applying a particular optimization will be beneficial, while at iterative compilation, a large number of different versions of the program are generated-executed by applying transformations and the fastest version is selected; iterative compilation provides good results, but requires extremely long compilation times.

The state of the art (SOA) hand/self-tuning libraries for linear algebra, such as ATLAS [16], OpenBLAS [17], Eigen [18], Intel_MKL [19], PHiPAC [20] and a few of the many noteworthy papers from the past such as [21] [22] [23] [24] for CPUs and [25] [26] for GPUs, do not give a theoretical solution; instead, they find the performance critical parameter values mostly by searching and by using heuristics and empirical techniques. The selection of the parameter values is a difficult and time consuming task for two reasons. First, many parameters have to be taken into account, such as the number of the levels of tiling, tile sizes, loop unroll depth, software pipelining strategies, register allocation, code generation, data reuse, loop transformations. Second, the optimum parameters for two slightly different architectures are different. Such a case is MMM algorithm, which is a major kernel in linear algebra and also the topic of this paper. In this paper the optimum scheduling parameters are found by decreasing the search space theoretically, for a wide range of CPU (including multi-cores) and GPU architectures.

A former MMM methodology for CPUs that support SIMD was introduced in [27]; the major contribution of [27] is that it addresses the major software and hardware parameters together, as one problem and not separately. This paper, extends [27] at four ways. First, the search space is decreased theoretically by applying static performance estimation. Second, this paper extends [27] to a wide range of computer architectures; this methodology has been extended to GPU architectures, to CPU architectures without SIMD unit and to CPU architectures different than those of the general purpose CPUs, e.g., microcontrollers, smaller processors with one level of data cache, processors with direct mapped data cache. Third, this methodology takes into account more hardware architecture parameters (the memories' latencies, the data cache

line size, the number, the latencies and the type of the CPU function units, the number of the load/store units) and software characteristics (number of matrix operations, number of addressing instructions). Fourth, the proposed methodology, due to the major contribution of number three above gives a smaller search space, a smaller code size and a smaller compilation time, as it does not test a large number of alternative schedules.

The proposed methodology is compared with the state of the art software libraries of ATLAS and Intel_MKL. Although a performance comparison with Intel_MKL is unfair, a detailed experimental analysis has been made as it is the fastest MMM library in the world for Intel general purpose processors. A performance comparison with Intel_MKL is unfair for two reasons. First, Intel_MKL developers have access to all the Intel processor architecture details which we do not, e.g., victim cache, hardware prefetchers; this is why Intel_MKL library is the fastest library on Intel processors only. Second, Intel_MKL loop kernels are written in assembly code while our method in C (assembly code is always more efficient); Intel developers write assembly code to deal with the low level transformations, e.g., register allocation, instruction selection and instruction scheduling. The proposed methodology lies at a higher level of abstraction and it is used to wide range of computer architectures. Implementing the proposed methodology in assembly code is beyond the scope of this paper and thus the low level transformations are applied by the target compiler (which is less efficient). The scope of this paper is not to provide the peak-performance MMM implementations, but to analytical give the architecture dependent high level transformation parameters (e.g., tile sizes) that achieve peak-performance. We strongly believe that if could modify the MKL library scheduling parameters according to the proposed methodology, an even higher performance would be achieved.

The proposed methodology is compared to the SOA libraries of ATLAS and Intel_MKL for CPUs and cuBLAS for GPUs. The evaluation is done by using Intel Xeon CPU E3-1241 v3, Pentium Intel i7-2600K, Valgrind tool [28], ARMv7-a on GEM5 simulator, PowerPC-440 on Xilinx FPGA Virtex-5, Nvidia GeForce GTX-580, Gem5 [29] and SimpleScalar simulator [30].

The remainder of this paper is organized as follows. In Sect. 2, the related work is given. The proposed methodology is presented in Sect. 3. In Sect. 4, the experimental results are presented while Sect. 5 is dedicated to conclusions.

2 Related Work

The problem of speeding up MMM is studied the last decates. A historical perspective is given in [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41]; these works present how hardware and software can work on scalable multi-processor systems for matrix algorithms.

ATLAS [16] [42] [43] [44] [45] is an implementation of a high performance software production/maintenance called Automated Empirical Optimization of Software (AEOS). In an AEOS enabled library, many different

ways of performing a given kernel operation are supplied, and timers are used to empirically determine which implementation is best for a given architectural platform. ATLAS uses BLAS implementations. The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. Intel Math Kernel Library (Intel MKL) [19] is a computing math library of highly optimized, extensively threaded routines for applications that require maximum performance. Intel_MKL library has been written by Intel and this is why it performs the best for Intel processors only. MKL kernels have been written in assembly for maximum performance. During the installation of ATLAS and Intel_MKL, on the one hand an extremely complex empirical tuning step is required, and on the other hand a large number of compiler options are used, both of which are not included in the scope of this paper. Although ATLAS is one of the SOA libraries for MMM algorithm, its techniques for supplying different implementations of kernel operations concerning memory management are empirical and hence it does not provide any methodology for it.

Regarding MMM implementations for one core, many related works exist such as [22,23,21,46–52]. In [22] BLIS is presented; BLIS is a framework for rapid instantiation of BLAS. In [23], BLIS extends the GotoBLAS approach to implement peak performance MMM implementations. In [21] a systematic analysis of the high-level issues affecting the design of high-performance matrix multiplication is given. Reference [24] gives a significant theoretical background on finding the optimum scheduling parameters, but it refers to specific CPUs architectures only. In [51], analytical models are presented for estimating the optimum tile size values assuming only fully associative caches, which in practice are very rare. The aforementioned works refer to specific CPU architectures only and find the scheduling parameters mostly by using empirical techniques.

Although loop-tiling is necessary to achieve high performance, the above works do not find the tile sizes and the number of levels of tiling, by taking into account the cache sizes, their associativities and the data arrays layouts, together as one problem; instead searching is applied. Let us give an example. According to ATLAS [16] (only cache size is taken into account), the size of three rectangular tiles (one for each matrix) must be smaller or equal to cache size; however, the elements of these tiles are not written in consecutive main memory locations and thus they do not use consecutive data cache locations; this means that having a set-associative cache, they cannot simultaneously fit in data cache due to the cache modulo effect. Moreover, even if the tiles elements are written in consecutive main memory locations (different data array layout), the three tiles cannot simultaneously fit in data cache if the cache is two-way associative or direct mapped. We will show that loop tiling is efficient only when cache size, cache associativity and data array layouts, are addressed together as one problem and not separately.

There are several works optimizing MMM for many cores [53–64]. The fastest implementations are given in [23] where MMM is parallelized on Intel Xeon Phi and on IBM Blue Gene/Q; an analysis is made on which loop is

going to be parallelized. The vast majority of previous works regarding multi-core architectures, deal with cluster architectures; they partition the MMM problem into many distributed memory computers (distributed memory refers to a multiple-processor computer system in which each processor has its own private memory). SRUMMA [64] describes one of the best parallel algorithm which is suitable for clusters and scalable shared memory systems. Although SRUMMA minimizes the communication contention between CPUs, it does not optimize the MMM problem for one CPU (it runs the *cblas_sgemm* ATLAS optimized routine). Furthermore, about half of the above works, use the Strassen's algorithm [65] to partition the MMM problem into many multi-core processors; Strassen's algorithm minimizes the number of the multiplication instructions sacrificing the number of add instructions and data locality. The MMM code for one core, is either given by Cilk tool [66] or by *cblas_sgemm* routine of ATLAS. At last, [67] and [68] show how shared caches can be utilized. All the above works, are empirical techniques and do not provide a theoretical model.

Regarding GPUs, several related works exist such as [25] [26] [69] [70] [71] [72] [73] [74] [75] [76]. Reference [26] show how to modify the MAGMA GEMM kernels in order to use more efficient the Fermi architecture. [69] presents a method for producing MMM kernels tuned only for a specific architecture, through a canonical process of heuristic autotuning, based on generation of multiple code variants and selecting the fastest ones through benchmarking. [71] provide implementations of Strassen's MMM algorithm as well as of Winograd's variant; they show that only for square matrices of very large sizes (16384×16384) achieve 33% speedup over *cblas_sgemm* (ATLAS routine for data of type float) and 21% over *cblas_dgemm* (ATLAS routine for data of type double). [73] presents an in-depth study to reveal interesting trade-offs between shared memory and the hardware-managed L1 data caches for MMM. [74] investigates different performance techniques such as tiling, memory coalescing, prefetching, and loop unrolling, in trying to evaluate which method is the most efficient. [75] provides theoretical analysis why performance drawbacks appear for specific problem sizes when using cache memories. Finally, in [76], different data arrays layouts are evaluated, such Z-Morton and X-Morton. All the above works, are empirical techniques and do not give a methodology.

In contrast to the proposed methodology, the above works find tile sizes mostly by searching, since they do not exploit all the h/w and the s/w constraints. However, if these constraints are fully exploited, the optimum solution can be found by enumerating only a small number of solutions; in this paper, tile sizes are given by inequalities which contain the cache sizes and cache associativities.

3 Proposed Methodology

In this paper a Matrix-Matrix Multiplication (MMM) methodology is presented where the sub-problems of finding the schedules with the minimum

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    for (k=0; k<P; k++)
      C[i][j] += A[i][k] * B[k][j];
```

Fig. 1 MMM unoptimized code

numbers of i) L1 data cache accesses, ii) L2 data cache accesses, iii) L3 data cache accesses, iv) main memory data accesses, v) addressing instructions, are addressed together as one problem and not separately. We find the scheduling parameters that achieve best performance by fully exploiting the software characteristics (production-consumption, data reuse and MMM parallelism) and the major hardware architecture parameters, i.e., the a) number of the cores, b) number of memories, c) size of each memory, d) number of registers, e) associativities of data cache memories, f) memories' latencies, e) SSE instruction latencies. For different hardware architecture parameters, different schedules for MMM are created.

It is well known that the search space, i.e., all different MMM implementations, is infinite and it cannot be searched. In this paper, we find the best schedules among these that exist in our search space. The search space being addressed consists of all the high level transformations that affect MMM performance including all different transformation orderings and all different transformation parameters (e.g., tile sizes); these are the number of levels of tiling, loop tiling for all the memories, register blocking, loop interchange, loop unroll, scalar replacement, data array layouts. The search space we use does not contain low level transformations, e.g., instruction scheduling to decrease the number of pipeline stalls (this is beyond the scope of this paper). Although, the low level transformations are not taken into account the search space being addressed is enormous and it is impractical to be searched, e.g., if the number of different tile sizes for each loop is 100, the number of all different schedules that apply loop tiling for L1 and L2 is $(6! \times 100^6) = 7.2 \times 10^{14}$ (two levels of tiling add 6 new loops); if we consider that the compilation time of each schedule is 1 sec and given that $1sec = 3.1 \times 10^{-8}$ years, the compilation time is very big; if we include all the above transformations, the number of the schedules becomes enormous. Given that the above transformations are strongly interdependent, the only way to decrease the search space is to be addressed together as one problem and not separately.

For the reminder of this paper, the three arrays names and sizes are that shown in Fig. 1, i.e., $C = C + A \times B$, where C, A and B are of size $N \times M$, $N \times P$ and $P \times M$, respectively.

MMM performance depends on the time needed to the i) data to be loaded/stored (C, A and B arrays), ii) matrix operations to be executed, iii) addressing instructions to be executed (integer instructions only), iv) instructions to be loaded from instruction cache.

Regarding (iv), the time needed for the instructions to be fetched from L1 instruction cache, is lower than the (i)-(iii) values above, since no instruction

cache conflicts occur; this is because the MMM code size is small and it always fits in L1 instruction cache. It is important to say that all the todays processors contain separate L1 data and instruction caches and thus we can assume that shared/unified L2/L3 caches, contain only data. Architectures with unified L1 caches are not discussed in this paper.

Eq. 1 and eq. 2 approximate the MMM execution time; eq. 1 holds for architectures that matrix operations and addressing operations are executed in parallel (we assume that the arrays are floating point numbers) and eq. 2 holds for architectures that do not (either no floating point Arithmetic Logic Unit (ALU) exists or the arrays are of type integer); in most architectures the load/store unit and the execution unit work in parallel.

$$T_{total} = max(T_{data}, T_{matrix-operations}, T_{addressing}) \tag{1}$$

$$T_{total} = max(T_{data}, T_{matrix-operations} + T_{addressing}) \tag{2}$$

Regarding the time needed to execute the matrix operations ($T_{matrix-operations}$), it is given by the following two equations, i.e., eq. 3 and eq. 4; eq. 3 is used in the case there is a separate multiplication unit working in parallel with the ALU and eq. 4 otherwise (the number of multiplications is larger than the number of additions). $T_{matrix-operations}$ is a constant number.

$$T_{matrix-operations} = \frac{Mul_{lat} \times (N \times M \times P)}{Num_{Mult}} \tag{3}$$

$$T_{matrix-operations} = \frac{Mul_{lat} \times (N \times M \times P)}{Num_{Mult}} + \frac{Add_{lat} \times (N \times M \times (P-1))}{Num_{ALU}} \tag{4}$$

where Mul_{lat}, Add_{lat}, Num_{Mult} and Num_{ALU} are the latencies and the numbers of the multiplication and ALU units, respectively.

Regarding addressing instructions, the $T_{addressing}$ value is not a constant number; it depends on the implementation/schedule. The number of addressing instructions is decreased when a) the number of the levels of tiling is decreased, b) the tile sizes are increased, c) more array references are assigned into available registers, d) loop unroll factor is increased.

Regarding load/store instructions, $T_{data} \succ T_{matrix-operations}$ in most cases, i.e., if the data do not fit in L1 data cache. On the contrast to the $T_{matrix-operations}$, T_{data} and $T_{addressing}$ are not constant numbers; they depend on the schedule used. Also, T_{data} and $T_{addressing}$ depend on each other; normally, by increasing T_{data} value, $T_{addressing}$ is decreased and vice versa, while the number of the matrix operations remains constant.

To summarize, given that $T_{matrix-operations}$ is a constant number and in most cases $T_{matrix-operations} \prec T_{addressing}$ and $T_{matrix-operations} \prec T_{data}$, MMM performance can be increased only by minimizing both T_{data} and $T_{addressing}$ values; given that T_{data} and $T_{addressing}$ are interdependent, high performance is achieved, only for both low T_{data} and $T_{addressing}$ values. This is because

the separate optimization of the T_{data} and $T_{addressing}$ values, gives different schedules which cannot coexist, as by refining one, degrading another.

T_{data} value is found by decreasing the search space theoretically according to the memory hierarchy architecture parameters. For different cache hierarchy architecture, a different equation that estimates T_{data} value is created, e.g., for one level of data cache architecture T_{data} value is given by eq.7. These equations give theoretically the tile sizes that achieve a minimum T_{data} value.

As far the $T_{addressing}$ value is concerned, it cannot be found theoretically. This is because the number of the addressing instructions highly depends on the target compiler and its optimizations, e.g., unroll factor values. Thus, we find all the schedules achieving a low T_{data} value and only these that achieve a low $T_{addressing}$ value are selected; all the schedules achieving a low T_{data} value are converted into assembly code (by using the target compiler) and their number of addressing instructions is measured. Then, all schedules achieving both low T_{data} and $T_{addressing}$ values are compiled and run to the target platform to find the fastest.

Instead of searching all different MMM schedules to find the best which is impractical because their number is enormous, only a small number is searched. The exploration space is decreased by orders of magnitude since we test only solutions that are close to the best; we test only these schedules achieving both low T_{data} and $T_{addressing}$ values. In this way, the compilation time is drastically decreased.

A different schedule is emerged for different types of CPUs, CPU parameters and input size. The reminder of this paper presents all these schedules. The proposed methodology for CPUs with one core is given in subsection 3.1, while the proposed methodology for CPUs with more than one cores is given in subsection 3.2. The proposed methodology for GPU architectures is given in subsection 3.3.

A different schedule is given according to the a) number and the type of the cores, b) number of cache memories, c) cache sizes, d) input sizes and e) whether an SIMD unit is supported or not (Fig. 2). In Fig. 2, 'S', 'M' and 'L' indicate small, medium and large input sizes, respectively. Also, the 3.1.1-3.1.6, 3.2.1, 3.2.2 and 3.3 values that are shown in Fig. 2 refer to the Subsect. 3.1.1 - Subsect. 3.1.6, Subsect.3.2.1, Subsect.3.2.2 and Subsect.3.3, respectively and they are given below.

3.1 CPUs with one core

A different schedule is given according to the a) number of data cache memories, b) cache sizes, c) input sizes and d) whether an SIMD unit is supported or not (Fig. 2). The Subsect. 3.1.1 - Subsect. 3.1.6 which are shown in Fig. 2, are given below.

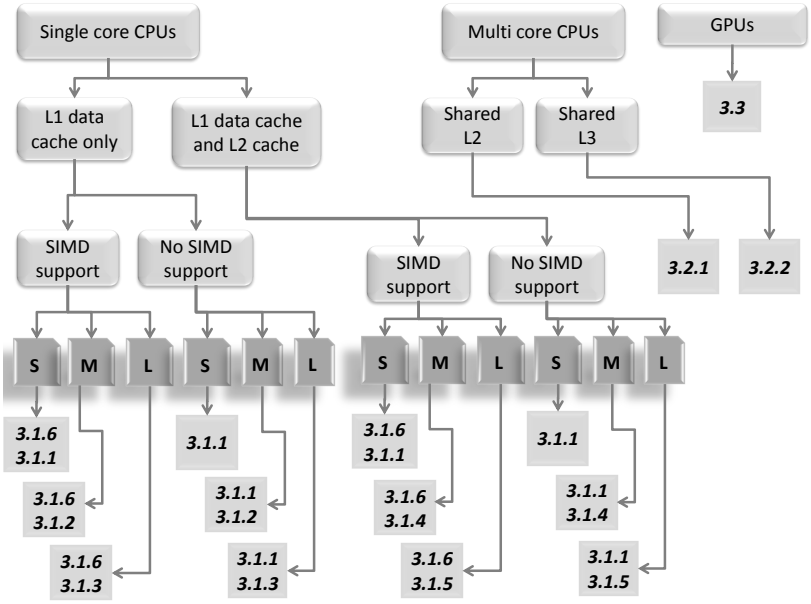

Fig. 2 All different MMM cases. The last nodes refer to the Subsections that provide the appropriate schedules. 'S', 'M' and 'L' indicate small, medium and large input sizes, respectively.

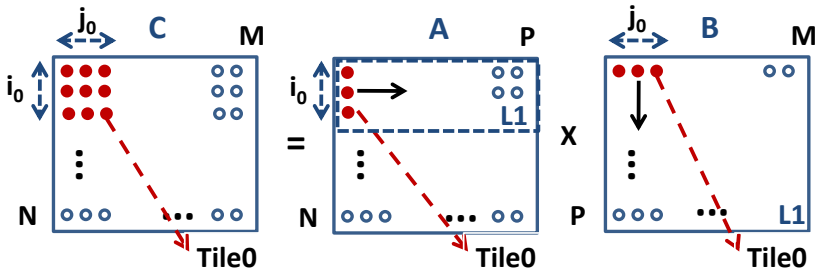

Fig. 3 MMM for small input sizes and CPU with L1 data cache and with/without L2 cache

3.1.1 Small input sizes and CPU with L1 data cache and with/without L2 cache

For small input sizes, i.e., if all the data of B, the data of $2 \times i_0$ rows of A and the data of $2 \times i_0$ rows of C fit in different ways of L1 data cache (ineq. 5), the scheduling given below is used (Fig. 3).

$$\frac{L1 \times (assoc - k1 - k2)}{assoc} \geq P \times M \times element\_size, \tag{5}$$

where $k1 = \lceil \frac{2 \times i_0 \times P \times element\_size}{L1/assoc} \rceil \leq \frac{assoc}{4}$, $k2 = \lceil \frac{2 \times i_0 \times M \times element\_size}{L1/assoc} \rceil \leq \frac{assoc}{4}$, $L1$ is the size of the L1 data cache memory in bytes, $element\_size$ is the size of the arrays elements in bytes and $assoc$ is the L1 associativity

($assoc \neq 1$). $P \times M$ is the size of the B array in elements. $k1$ is an integer and it gives the number of L1 data cache lines with identical L1 addresses used for $2 \times i_0$ rows of A; $k2$ is an integer and it gives the number of L1 data cache lines with identical L1 addresses used for $2 \times i_0$ rows of C; for the reminder of this paper we will more freely say that we use k1 cache ways for A, k2 ways for C and $(assoc - k1 - k2)$ cache ways for B (in other words A, B and C are written in separate data cache ways). In the case that $P \times M \gg 2 \times i_0 \times P + 2 \times i_0 \times M$ or in the case that a two-way set associative cache exists, 1 cache way is used for both A and C. In the case that $assoc = 1$, a slightly different schedule is given at the last paragraph of Subsect. 3.1.3.

The optimum production-consumption (when an intermediate result is produced it is directly consumed-used) of array C and the sub-optimum data reuse of array A have been selected by splitting the arrays into tiles according to the number of the registers, eq. 6 (Fig. 3). All different i_0, j_0 combinations satisfying ineq. 6 give a feasible solution.

$$RF_{FP} \geq i_0 \times j_0 + i_0 + j_0 \tag{6}$$

where RF_{FP} is the number of the available floating point registers. In the case that C,A and B contain integer numbers, ineq. 6 contains the addressing variables and the loop iterators.

We use $i_0 \times j_0$ registers for C, i_0 for A and j_0 for B (Fig. 3). We assign registers across i, j iterators (Fig. 1) and not across k iterator because k is the innermost one (it is changes its value in each iteration and thus no data reuse is achieved).

The schedule is shown in Fig. 3. First, the first i_0 elements of the first column of A are multiplied by the first j_0 elements of the first row of B. Then, the first i_0 elements of the second column of A are multiplied by the first j_0 elements of the second row of B etc. This is repeated until the first i_0 rows of A have been multiplied by the first j_0 columns of B; then the same i_0 rows of A as above, are multiplied by the next j_0 columns of B etc.

Given that each row of A is multiplied by all columns of B, both A and B are reused M and N times, respectively; thus, they have to remain in L1 data cache. To do this, the cache lines of A and B must be written in L1 without conflict with each other and also with C. $2 \times i_0$ rows of A and C have to fit in L1, the current processed i_0 rows and the next ones, for two reasons. First, except from the first i_0 rows of A and C, also the second i_0 rows must be loaded in L1, without conflict with B. Second, when the third i_0 rows of A are multiplied by B, the L1 cache lines containing the first i_0 rows are replaced by the third i_0 rows ones according to the LRU cache replacement policy, without conflict with the B ones. This is achieved by storing the rows of A, C and the columns of B in consecutive main memory locations and by using $(k1 \times \frac{L1}{assoc})$ L1 memory size for A, $(k2 \times \frac{L1}{assoc})$ for C and $((assoc - k1 - k2) \times \frac{L1}{assoc})$ for B (ineq. 5). We can more freely say that this is equivalent to using $k1$ cache ways for A, $k2$ cache ways for C and $(assoc - k1 - k2)$ cache ways for B. An empty cache line is always granted for each different modulo (with respect to

Table 1 Number of data accesses in memory hierarchy

Subsection 3.1.1 case Ineq.5 holds			
No optimization			
	C	A	B
L1	$N \times M$	$N \times P \times M$	$P \times M \times N$
DDR	$N \times M$	$N \times P$	$P \times M$
Schedule in Subsection 3.1.1			
	C	A	B
L1	$N \times M$	$\frac{N \times P \times M}{j_0}$	$\frac{P \times M \times N}{i_0}$
DDR	$N \times M$	$N \times P$	$P \times M$
Subsection 3.1.2 case Ineq.6 holds			
No optimization			
	C	A	B
L1	$N \times M$	$N \times P \times M$	$P \times M \times N$
DDR	$N \times M$	$N \times P$	$P \times M \times N$
Schedule in Subsection 3.1.2			
	C	A	B
L1	$N \times M$	$\frac{N \times P \times M}{j_0}$	$\frac{P \times M \times N}{i_0}$
DDR	$N \times M$	$N \times P$	$\frac{P \times M \times N}{II}$
Subsection 3.1.3 case Ineq.6 does not hold			
No optimization			
	C	A	B
L1	$N \times M$	$N \times P \times M$	$P \times M \times N$
DDR	$N \times M$	$N \times P \times M$	$P \times M \times N$
Schedule in Subsection 3.1.3			
	C	A	B
L1	$\frac{N \times M \times P}{KK}$	$\frac{N \times P \times M}{j_0}$	$\frac{P \times M \times N}{i_0}$
DDR	$\frac{N \times M \times P}{KK}$	$N \times P$	$\frac{P \times M \times N}{II}$
Subsection 3.1.4 case Ineq.7 holds			
No optimization			
	C	A	B
L1	$N \times M$	$N \times P \times M$	$P \times M \times N$
L2	$N \times M$	$N \times P$	$P \times M \times N$
DDR	$N \times M$	$N \times P$	$P \times M$
Schedule in Subsection 3.1.4			
	C	A	B
L1	$N \times M$	$\frac{N \times P \times M}{j_0}$	$\frac{P \times M \times N}{i_0}$
L2	$N \times M$	$\frac{N \times P \times M}{JJ}$	$P \times M$
DDR	$N \times M$	$N \times P$	$P \times M$
Subsection 3.1.5 case Ineq.7 does not hold			
No optimization			
	C	A	B
L1	$N \times M$	$N \times P \times M$	$P \times M \times N$
L2	$N \times M$	$N \times P \times M$	$P \times M \times N$
DDR	$N \times M$	$N \times P \times M$	$P \times M \times N$
Schedule in Subsection 3.1.5			
	C	A	B
L1	$\frac{N \times M \times P}{KK}$	$\frac{N \times P \times M}{j_0}$	$\frac{P \times M \times N}{i_0}$
L2	$\frac{N \times M \times P}{KK}$	$\frac{N \times P \times M}{j_0}$	$\frac{P \times M \times N}{II}$
DDR	$\frac{N \times M \times P}{KK}$	$N \times P$	$\frac{P \times M \times N}{II}$

the size of the cache) of A, C and B memory addresses. It is important to say that if we use $L1 \geq (P \times M + 2 \times i_0 \times P + 2 \times i_0 \times M) \times element\_size$ instead of ineq. 5, the number of L1 misses will be much larger because A, B and C cache lines would conflict with each other.

The time needed for the array elements to be loaded/stored, is approximated by eq. 7 and Table 1.

$$T_{data} = max(\frac{L1_{load\_lat} \times L1_{loads} + L1_{store\_lat} \times L1_{stores}}{L1_{ports}},$$
$$\frac{DDR_{load\_lat}}{line_{L1}} \times DDR_{loads} + DDR_{store\_lat} \times \frac{DDR_{stores} \times \lceil j_0/line_{L1} \rceil}{j_0}) \quad (7)$$

where $L1_{load\_lat}$, DDR_{load_lat}, $L1_{store\_lat}$, DDR_{store_lat} are the L1 and DDR load and store latencies, respectively. $L1_{loads}$, DDR_{loads}, $L1_{stores}$, DDR_{stores}, are the numbers of loads and stores occur for each memory and they are shown in Table 1. $line_{L1}$ is the number of elements each L1 cache line contains and $L1_{ports}$ is the number of L1 load/store ports. Without loss of generality, in eq. 7, we assume a memory architecture that only one L1 cache line is replaced at a time; if more than one cache lines are replaced in parallel, then eq. 7 is changed accordingly.

Regarding the number of DDR writes, $\lceil j_0/line_{L1} \rceil$ L1 data cache lines are written to main memory for each j_0 elements of C. For data cache architectures where reads and writes are executed in parallel, eq. 7 is changed accordingly.

From eq. 7 and Table 1, we can approximate the T_{data} value. Supposing that L1 and DDR load/store latencies are 1 and 50, respectively and also that $L1_{ports} = 1$ and $line_{L1} = 4$, eq. 7 and Table 1 give

$$T_{data} = max(2 \times N \times M + \frac{N \times P \times M}{i_0} + \frac{P \times M \times N}{j_0},$$
$$\frac{50}{4} \times (N \times M + N \times P + P \times M) + 50 \times \frac{N \times M \times \lceil \frac{j_0}{4} \rceil}{j_0}) \quad (8)$$

Regarding DDR access time, it is minimized when j_0 is a multiple of cache line size, i.e., 4. Regarding L1 data cache access time, it is minimized when $(\frac{1}{i_0} + \frac{1}{j_0})$ is minimized (i_0, j_0 are found according to ineq. 6 and RF_{FP} is a power of 2). For $RF_{FP} = 8$ or $RF_{FP} = 16$, the above equation is minimized when $i_0 == j_0$. Let us consider that there are 16 floating point registers. If $i_0 == j_0$, then $i_0 = j_0 = 3$ and the number of L1 data cache accesses is $N \times M + \frac{2NMP}{3}$, while if $i_0 = 1$ and $j_0 = 14$, the number of L1 data cache accesses is $N \times M + \frac{15NMP}{14} \succ N \times M + \frac{2NMP}{3}$. Also, if $RF_{FP} = 8$, then $i_0 = j_0 = 2$ gives the minimum L1 data cache access time. However, in the case that $RF_{FP} = 32$, we cannot find a good $i_0 = j_0$ solution, since several registers are wasted, i.e., if $i_0 = j_0 = 4$ then 16, 4 and 4 registers are used for C, A and B, respectively, which means that only 24/32 registers are used; in this case, we do not fully utilize the RF size and thus a solution different than $i_0 = j_0$ is preferred, e.g., $i_0 = 5$ and $j_0 = 4$. On the other hand, if we select $i_0 = j_0 = 5$, then we need 35 register which are more than 32.

However, the best performance is not always achieved by minimizing T_{data} value; although $i_0 == j_0$ case achieves a lower number of data accesses than $j_0 \succ i_0$ case (for the most register file sizes), it achieves a larger number of

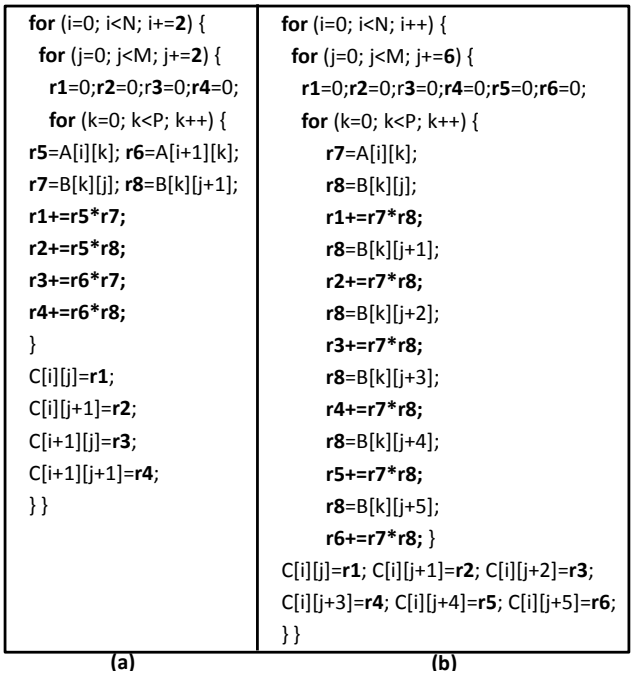

Fig. 4 MMM optimized code for $RF_{FP} = 8$. In (a) $i_0 = j_0 = 2$, while in (b) $i_0 = 1$ and $j_0 = 6$. (a) achieves a lower T_{data} value but (b) achieves a lower $T_{addressing}$ value

addressing instructions; this is because in the first case, more array addresses per iteration, are computed (Fig. 4).

The above schedule achieves the minimum number of data accesses (for the most register file sizes). In this case, C array is loaded and stored once from L1 data cache, while A and B are loaded N/i_0 and M/j_0, respectively (for square matrices, N^2 stores and $N^2 + N^3/i_0 + N^3/j_0$ loads occur). The row-column way of multiplying is the best. If we use another schedule, e.g., loop interchange transformation is applied and the iterators are k, i, j, instead of i, j, k and thus we use $i_0 \times j_0$, i_0 and j_0 registers for A, C and B, respectively, then N^3/i_0 stores and $N^2 + N^3/i_0 + N^3/j_0$ loads occur, which are more than those in the previous case. A larger number of data accesses occurs because C is always accessed twice (C array is both loaded and stored); thus, it is more efficient to access C array once and A, B arrays more times, than the opposite.

3.1.2 Medium input sizes with L1 data cache only

For medium input sizes where ineq. 9 holds, another schedule is used (Fig. 5). If all the data of a Tile1 of A and a Tile1 of B fit in separate ways of L1 cache (ineq. 9), the scheduling given below is used.

$$\frac{L1 \times (assoc - k)}{assoc} \geq II \times P \times element\_size \tag{9}$$

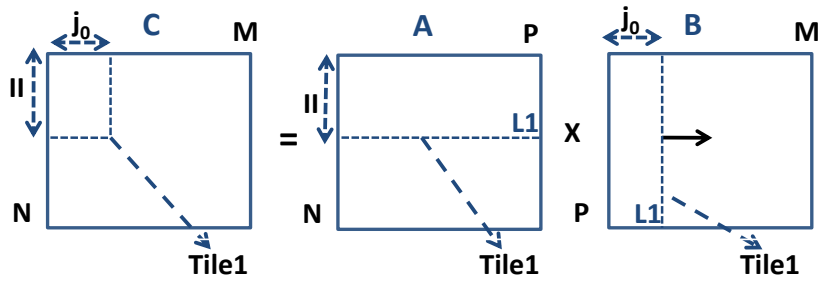

Fig. 5 MMM for medium input sizes and CPUs with L1 data cache only

where $k = \lceil \frac{P \times j_0 \times element\_size}{L1/assoc} \rceil \leq \frac{assoc}{2}$, $L1$ is the size of the L1 cache, $assoc$ is the L1 associativity ($assoc \neq 1$) and $element\_size$ is the size of the arrays elements in bytes.

k is an integer and it gives the number of L1 data cache lines with identical L1 addresses used for Tile1 of B; we use k cache ways for B and $assoc - k$ cache ways for A (in other words A and B are written in separate data cache ways).

In order to the Tile1 tiles of A and B remain in L1 data cache, the cache lines of Tile1 of A must be written in L1 without conflict with the Tile1 of B ones. This is achieved by storing the Tile1 of A and the Tile1 of B in consecutive main memory locations and by using $(k \times \frac{L1}{assoc})$ L1 memory size for B and $((assoc - k) \times \frac{L1}{assoc})$ L1 memory size for A (ineq. 9). We can more freely say that this is equivalent to using k cache ways for B and $(assoc - k)$ cache ways for A. An empty cache line is always granted for each different modulo (with respect to the size of the cache) of A and B memory addresses. We do not need any empty space for C because a) Tile1 of C size is much smaller than the other Tile1 tiles (normally, $P \gg II \succ j_0$), b) each element of C is stored just once into memory (no data reuse) and c) C is stored into main memory infrequently; thus, the number of conflicts due to C can be neglected (victim cache if exists, eliminates the misses of C). However, if P is comparable to II, then an additional cache way must be used for C.

Ineq. 9 holds only when Tile1 of A and B are written in consecutive main memory locations. Given that the arrays are written row-wise in main memory, the elements of Tile1 of A are written in consecutive main memory locations, but the elements of Tile1 of B are not. Thus, the data layout of B is changed from row-wise to tile-wise, i.e., all elements are written in main memory just as they are fetched; first, the first j_0 elements of the first row of B are written to main memory, then the first j_0 elements of the second row of B etc.

It is important to say that if we use $L1 \geq (II \times P + P \times j_0) \times element\_size$ instead of ineq. 9, the number of L1 misses will be much larger because A and B cache lines would conflict with each other.

The scheduling follows. First, the first i_0 rows of A are multiplied by the first j_0 columns of B exactly as in the previous subsection. Then the next i_0 rows of A are multiplied by the same columns of B as above. This is repeated

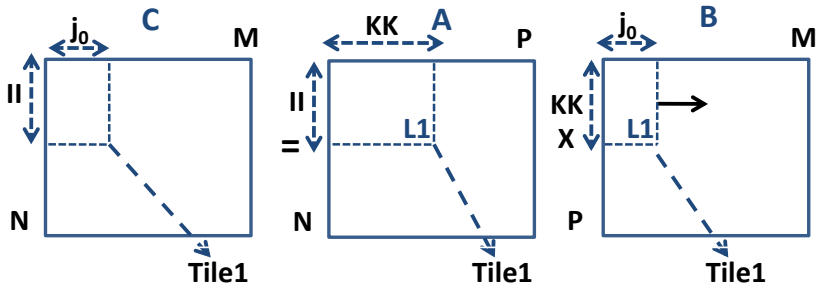

Fig. 6 MMM for large input sizes and CPUs with L1 data cache only

until all the rows of Tile1 of A are multiplied by the first j_0 columns of B. After the first Tile1 of A has been multiplied by the first Tile1 of B, the first Tile1 of A is multiplied by the second Tile1 of B etc. In this way, data reuse is achieved on both A and B arrays.

The time needed for the arrays elements to be loaded/stored, is approximated by eq. 7 and Table 1. Supposing that L1 and DDR load/store latencies are 1 and 50, respectively and also that $L1_{ports} = 1$ and $line_{L1} = 4$, eq. 7 and Table 1 give

$$T_{data} = max(2 \times N \times M + \frac{N \times P \times M}{i_0} + \frac{P \times M \times N}{j_0},$$
$$\frac{50}{4} \times (N \times M + N \times P + \frac{P \times M \times N}{II}) + 50 \times \frac{N \times M \times \lceil \frac{j_0}{4} \rceil}{j_0}) \quad (10)$$

Regarding DDR access time, it is minimized when II is maximized and j_0 is a multiple of cache line size, i.e., 4; also II is maximized when $j_0 = 1$ (II and j_0 are interdependent); there is a trade-off. Regarding L1 data cache access time, it is minimized when $(\frac{1}{i_0} + \frac{1}{j_0})$ is minimized (i_0, j_0 are found according to ineq. 6).

3.1.3 Large input sizes and CPUs with L1 data cache only

For large input sizes, where ineq. 9 does not hold, the scheduling given below is used (Fig. 6).

In this case, i_0 rows of A and j_0 columns of B do not fit in L1; ineq. 9 cannot give $j_0 \succ 1$ and $i_0 \succ 1$. Thus, P dimension is also tiled and Tile1 tiles become even smaller; Tile1 tiles contain i_0 sub-rows of A and j_0 sub-columns of B; the Tile1 tiles of A and B are of size $II \times KK$ and $KK \times j_0$, respectively. KK is selected to be as large as possible since C array is both loaded and stored, KK times from main memory.

Now, instead of multiplying rows of A by columns of B, sub-rows of size KK are multiplied by sub-columns of B. The largest II integer value for $KK = P/2$ is selected that ineq. 11 holds. If ineq. 11, still cannot give an II value satisfying ineq. 6, $KK = P/3$ is selected etc.

$$\frac{L1 \times (assoc - k)}{assoc} \geq II \times KK \times element\_size \quad (11)$$

where $k = \lceil \frac{j_0 \times KK \times element\_size}{L1/assoc} \rceil \leq \frac{assoc}{2}$, $KK = \frac{P}{1}, \frac{P}{2}, ..., \frac{P}{n}$ and n is positive integer ($n \geq 1$).

II sub-rows of A and j_0 sub-columns of B of size KK, have to fit in separate ways of L1 data cache. It is important to say that ineq. 11 holds only when the tile elements of A and B are written in consecutive main memory locations; otherwise, the tile sub-rows/sub-columns will conflict with each other due to the cache modulo effect. As it was explained in the previous Subsection, we do not need any empty space for C. However, if the size of Tile1 of C is comparable to the others, then cache size which equals to one cache way must be granded for C.

Regarding the data layout of A, when P dimension is tiled, the data layout of A is changed from row-wise to tile-wise; A elements are written into memory just as they are fetched. If the data layout of A is not changed, ineq. 11 cannot give a minimum number of cache conflicts since the sub-rows of A will conflict with each other. The same holds for B. The data array layout of B is changed from row-wise to tile-wise.

The scheduling follows. The multiplication between two Tile1 tiles is exactly the same as in the previous case (Subsection 3.1.1). First, the first Tile1 of A is multiplied by all Tile1 tiles of the first Tile1 block row of B. Then, the second Tile1 of the first Tile1 block column of A is multiplied by all the Tile1 tiles of the first Tile1 block row of B. Then, the second Tile1 block column of A is multiplied by the second Tile1 block row of B etc.

The time needed for the array elements to be loaded/stored, is approximated by eq. 7 and Table 1. Supposing that L1 and DDR load/store latencies are 1 and 50, respectively and also that $L1_{ports} = 1$ and $line_{L1} = 4$, eq. 7 and Table 1 give

$$T_{data} = max(\frac{2 \times N \times M \times P}{KK} + \frac{N \times P \times M}{i_0} + \frac{P \times M \times N}{j_0},$$
$$\frac{50}{4} \times (\frac{N \times M \times P}{KK} + N \times P + \frac{P \times M \times N}{II}) + 50 \times \frac{\frac{N \times M \times P}{KK} \times \lceil \frac{j_0}{4} \rceil}{j_0}) \quad (12)$$

Regarding DDR access time, it is minimized when KK and II are maximized and when j_0 is a multiple of cache line size, i.e., 4. T_{data} highly depends on the KK and II values which are the largest possible according to the L1 data cache size.

In the case of direct mapped data cache, both A and B tiles compete with each other for the same L1 addresses. Given that both tiles cannot remain in L1 due to the cache modulo effect, we select Tile1 of A to be many times larger than Tile1 of B, i.e., $II \gg j_0$. In this way, the main part of Tile1 of A remain in L1 and the number of L1 misses is kept low.

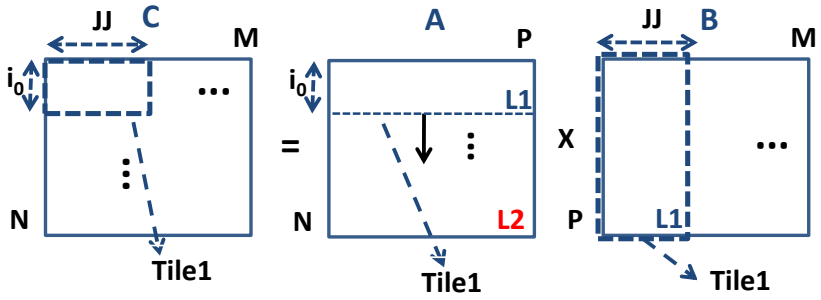

Fig. 7 MMM for medium input sizes and CPUs with L2 and L1 data cache

3.1.4 Medium input sizes and CPUs with L2 and L1 data cache

If all the data of A and the Tile1 of B/C fit in separate ways of L2 cache (ineq. 13), the scheduling given below is used (Fig. 7).

$$\frac{L2 \times (assoc - 1)}{assoc} \geq N \times P \times element\_size \tag{13}$$

where $L2$ is the size of the L2 cache, $assoc$ is the L2 associativity and $element\_size$ is the size of the arrays elements in bytes (e.g., $element\_size = 4$ for floating point numbers).

Given that $N \gg JJ$ for most cases, the size of A is much larger than the size of Tile1 of B and Tile1 of C, and thus $((assoc - 1) \times \frac{L2}{assoc})$ and $(\frac{L2}{assoc})$ L2 size is needed for A and B-C arrays, respectively.

Regarding L1 data cache, $2 \times i_0$ rows of A and JJ columns of B have to fit in separate ways of L1 (ineq. 14).

$$\frac{L1 \times (assoc - k)}{assoc} \geq P \times JJ \times element\_size, \tag{14}$$

where $k = \lceil \frac{2 \times i_0 \times P \times element\_size}{L1/assoc} \rceil \leq \frac{assoc}{2}$, $L1$ is the size of the L1 data cache memory in bytes, $element\_size$ is the size of the arrays elements in bytes and $assoc$ is the L1 associativity ($assoc \neq 1$).

The scheduling follows. First, the first Tile1 of A is multiplied by the first Tile1 of B. The multiplication between two Tile1 tiles is the same as in Subsect. 3.1.1. Then, the second Tile1 of A is multiplied by the same Tile1 tiles of B as above etc. We select the whole A to fit in L2, since having a Tile1 of B in L1 data cache, its elements need to be multiplied by as many rows of A as possible before they are spilled to upper level memories.

The time needed for the array elements to be loaded/stored, is approximated by eq. 19 and Table 1.

$$T_{data} = max(\frac{L1_{load\_lat} \times L1_{loads} + L1_{store\_lat} \times L1_{stores}}{L1_{ports}},$$

$$\frac{L2_{load\_lat}}{line_{L1}} \times L2_{loads} + L2_{store\_lat} \times \frac{L2_{stores} \times \lceil j_0/line_{L1} \rceil}{j_0},$$
$$\frac{DDR_{load\_lat}}{line_{L2}} \times DDR_{loads} + DDR_{store\_lat} \times \frac{DDR_{stores} \times \lceil j_0/line_{L2} \rceil}{j_0}) \quad (15)$$

where $L1_{load\_lat}$, $L2_{load\_lat}$, DDR_{load_lat}, $L1_{store\_lat}$, $L2_{store\_lat}$, DDR_{store_lat} are the L1,L2 and DDR load and store latencies, respectively. $L1_{loads}$, $L2_{loads}$, DDR_{loads}, $L1_{stores}$, $L2_{stores}$, DDR_{stores}, are the numbers of loads and stores occur for each memory, according to Table 1. $line_{L1}/line_{L2}$ are the numbers of elements each L1/L2 cache line contains and $L1_{ports}$ is the number of L1 load/store ports. In eq. 7, we assume that only one L1/L2 cache line is replaced at the time; if more than one cache lines are replaced concurrently, e.g., two lines, then eq. 7 is changed accordingly.

Supposing that L1, L2 and DDR load/store latencies are 1, 4 and 50, respectively and also that $L1_{ports} = 1$, $line_{L1} = 4$, $line_{L2} = 4$, eq. 7 and Table 1 give

$$T_{data} = max(2 \times N \times M + \frac{N \times M \times P}{i_0} + \frac{N \times M \times P}{j_0},$$
$$N \times M + \frac{N \times M \times P}{JJ} + P \times M + 4 \times \frac{N \times M \times \lceil \frac{j_0}{4} \rceil}{j_0},$$
$$\frac{50}{4}(N \times M + N \times P + P \times M) + 50 \times \frac{N \times M \times \lceil \frac{j_0}{4} \rceil}{j_0}) \quad (16)$$

Regarding DDR access time, it is minimized when j_0 is a multiple of L2 cache line size, i.e., 4. Regarding L2 data cache access time, it is minimized when JJ is maximized and j_0 is a multiple of L1 cache line size, i.e., 4. JJ is maximum according to the L1 data cache size. Regarding L1 data cache access time, it is minimized when $(\frac{1}{i_0} + \frac{1}{j_0})$ is minimized.

3.1.5 Large input sizes and CPUs with L2 and L1 data cache

For large input sizes, where ineq. 13 does not hold, the scheduling given below is used (Fig. 8).

In this case, ineq. 14 cannot give $JJ \succ 1$ and $i_0 \succ 1$. Thus, P dimension is also tiled as in Subsect. 3.1.3 and Tile1 tiles become even smaller.

Thus, instead of multiplying rows of A by columns of B, sub-rows of size KK are multiplied by sub-columns of size KK. The Tile1 of A becomes of size $i_0 \times KK$ and the Tile1 of B becomes of size $KK \times j_0$; the largest j_0 integer value for $KK = P/2$ is selected that ineq. 17 holds. If ineq. 17, still cannot give a j_0 value satisfying ineq. 6, $KK = P/3$ is selected and etc.

$$\frac{L1 \times (assoc - k)}{assoc} \geq KK \times j_0 \times element\_size \quad (17)$$

where $k = \lceil \frac{2 \times i_0 \times KK \times element\_size}{L1/assoc} \rceil \leq \frac{assoc}{2}$, $KK = \frac{P}{1}, \frac{P}{2}, ..., \frac{P}{n}$ and n is positive integer ($n \geq 1$).

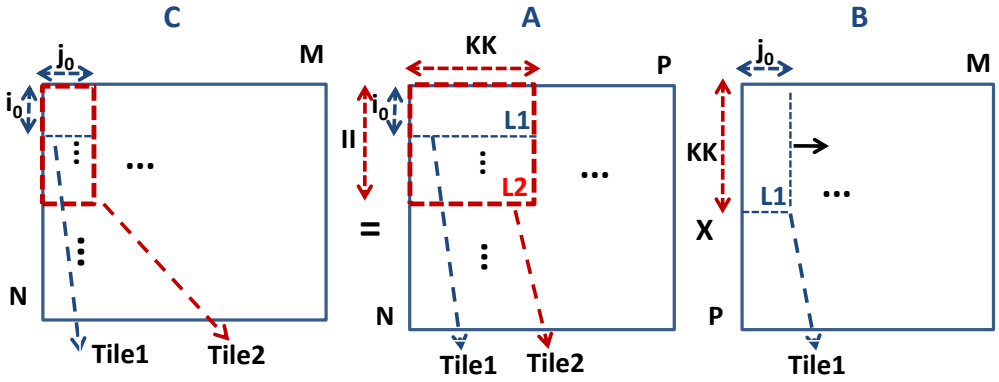

Fig. 8 MMM for large input sizes and CPUs with L2 and L1 data cache

$2 \times i_0$ sub-rows of A and j_0 sub-columns of B of size KK, have to fit in separate ways of L1 data cache. It is important to say that ineq. 17 holds only when the tiles of A and B are written in consecutive main memory locations (tile-wise); otherwise, the tiles sub-rows/sub-columns will conflict with each other due to the cache modulo effect.

To efficiently use the L2 cache, the array A is further partitioned into $Tile2$ tiles. A $Tile2$ tile of A (size of $II \times KK$), a Tile1 tile of B (size of $KK \times j_0$) and a Tile2 of C (size of $II \times j_0$), have to fit in L2 cache (ineq. 18). Array A uses $assoc - 1$ L2 ways while B-C arrays use only one L2 way. This is because the sum of Tile2 of C and Tile1 of B, is of smaller size than one L2 way ($II \gg j_0$); moreover, C and B tiles do not achieve data reuse in L2 and thus there is no need to remain in L2 (Tile1 of B is reused in L1 data cache not in L2).

$$\frac{L2 \times (assoc - 1)}{assoc} \geq II \times KK \times element\_size \tag{18}$$

The scheduling follows. First, the first Tile1 of the first Tile2 of A is multiplied by the first Tile1 of the first Tile1 block row of B. Then, the second Tile1 of the first Tile2 of A is multiplied by the same Tile1 of B as above etc. After the first Tile2 of A has been multiplied by the first Tile1 of B, it is multiplied by the remaining Tile1 tiles of the first Tile1 block row of B. Then, the second Tile2 of the first Tile2 block column of A is multiplied by the all Tile1 tiles of the first Tile1 block row of B etc. The procedure ends when all Tile2 block columns of A have been multiplied by all Tile1 block rows of B.

We select a big Tile2 of A to fit in L2, since a Tile1 of B which resides in L1 data cache, is multiplied by all rows of A; thus we multiply Tile1 of B with as many rows of A as possible before they are spilled to the upper level memory.

The time needed for the array elements to be loaded/stored, is approximated by eq. 19 and Table 1. Supposing that L1, L2 and DDR load/store latencies are 1, 4 and 50, respectively and also that $L1_{ports} = 1$, $line_{L1} = 4$, $line_{L2} = 4$, eq. 7 and Table 1 give

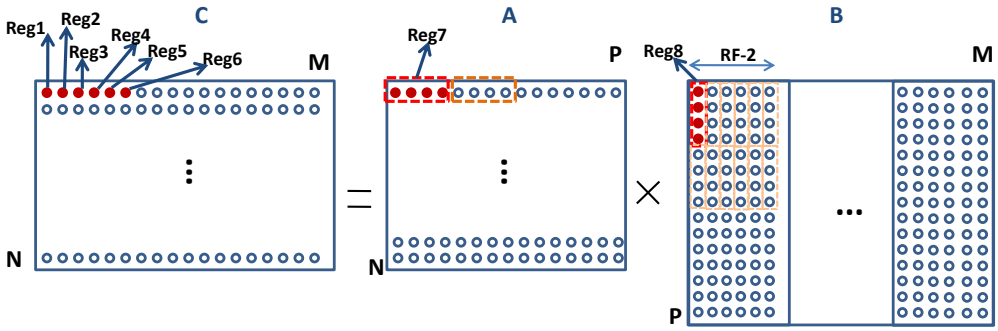

Fig. 9 MMM for CPUs with SIMD

$$T_{data} = max(\frac{2 \times N \times M}{KK} + \frac{N \times M \times P}{i_0} + \frac{N \times M \times P}{j_0},$$
$$\frac{N \times M \times P}{KK} + \frac{N \times M \times P}{j_0} + \frac{P \times M \times N}{II} + 4 \times \frac{\frac{N \times M \times P}{KK} \times \lceil \frac{j_0}{4} \rceil}{j_0},$$
$$\frac{50}{4}(\frac{N \times M \times P}{KK} + N \times P + \frac{N \times M \times P}{II}) + 50 \times \frac{\frac{N \times M \times P}{KK} \times \lceil \frac{j_0}{4} \rceil}{j_0}) \quad (19)$$

3.1.6 CPUs with SIMD

In this case, a scheduling similar to that explained in Subsect. 3.1.1 is used. The optimum production-consumption (when an intermediate result is produced it is directly consumed-used) of array C and the sub-optimum data reuse of array A have been selected by splitting the arrays into tiles according to the number of XMM/YMM registers (eq. 20).

$$RF = p + 1 + 1 \quad (20)$$

where RF is the number of the XMM/YMM registers and p is the number of the registers used for C array. Thus, we assign p registers for C and 1 register each for A and B.

Regarding T_{data} value, the best schedule here is similar to than given in Subsect. 3.1.1, i.e., 2×2, 2 and 2 registers are used for C, A and B, when $RF = 8$ and 3×3, 3 and 3 registers are used for C, A and B, when $RF = 16$. However, the schedule according to ineq. 20 is faster on SIMD architectures since the lower number of addressing instructions has a larger effect on performance. As the p value increases, the number of SSE instructions decreases (it is explained below). We have evaluated both solutions in a large number of different architectures and the first (that of Fig. 9) is the fastest at all architectures.

The illustration example consists of the scenario that there are 8 XMM registers (XMM0:XMM7 of 16 bytes each) and the arrays contain floating point

data (4 byte elements). The first 4 elements of the first row of A ($A(0, 0:3)$) and the first four elements of the first column of B ($B(0:3, 0)$) are loaded from memory and they are assigned into XMM0 and XMM1 registers respectively (the elements of B have been written into main memory tile-wise, i.e., just as they are fetched). XMM0 is multiplied by XMM1 and the result is stored into XMM2 register (Fig. 4). Then, the next four elements of B ($B(0:3, 1)$), are loaded into XMM1 register again; XMM0 is multiplied by XMM1 and the result is stored into XMM3 register (Fig. 9, Fig. 4). The XMM0 is multiplied by XMM1 for 6 times and the XMM2:XMM7 registers contain the multiplication intermediate results of the C array. Then, the next four elements of A ($A(0, 4:7)$) are loaded into XMM0 which is multiplied by XMM1 for 6 times, as above (Fig. 4); the intermediate results in XMM2:XMM7 registers, are always produced and consumed. When the 1st row of A has been multiplied by the first 6 columns of B, the four values of each one of XMM2:XMM7 registers are added and they are stored into main memory (C array), e.g., the sum of the four XMM2 values, is $C(0, 0)$.

The above procedure continues until all the rows of A have been multiplied by all the columns of B. There are several ways to add the XMM2:XMM7 data; three of them are shown in Fig. 10, where 4 XMM registers are used to store the data of C, i.e., XMM1, XMM2, XMM3 and XMM4 (the SSE instructions' latencies are taken into account here). The first one (Fig. 10-a) sums the four 32-bit values of each XMM register and the results of the four registers are packed in one which is stored into memory (the four values are stored into memory using one SSE instruction). The second one, sums the four 32-bit values of each XMM register and then each 32-bit value is stored into memory separately (without packing). For most SIMD architectures, the second (Fig. 10-b) is faster than the first one, because the store and add operations can be executed in parallel (the first one has a larger critical path). The third one (Fig. 10-c), unpacks the 32-bit values of the four registers and packs them into new ones in order to add elements of different registers. For most SIMD architectures, the third is faster than the other two ones, because unpacking and shuffle operations usually have smaller latency and throughput values than slow hadd operations.

By using SSE instructions, eq.1 changes into eq. 21. KK is not shown in Fig. 9; KK is the tile size across dimension P and for large input sizes $KK \prec P$ (Fig. 8). As the KK value decreases, the number of SSE instructions increases according to the following equation (code shown in Fig. 10 is executed more times).

$$T_{total} = max(T_{data}, T_{matrix-operations} + c \times \frac{N \times M \times \frac{P}{KK}}{p}, T_{addressing}) \quad (21)$$

where c is the sum of the SSE instruction latencies shown in Fig. 10.

Thus, as the p value increases, the number of SSE instructions decreases.

```
xmm1=_mm_hadd_ps(xmm1, xmm1);
xmm1=_mm_hadd_ps(xmm1, xmm1);
xmm2=_mm_hadd_ps(xmm2, xmm2);
xmm2=_mm_hadd_ps(xmm2, xmm2);
xmm3=_mm_hadd_ps(xmm3, xmm3);
xmm3=_mm_hadd_ps(xmm3, xmm3);
xmm4=_mm_hadd_ps(xmm4, xmm4);
xmm4=_mm_hadd_ps(xmm4, xmm4);

xmm1=_mm_unpacklo_ps(xmm1,xmm2);
xmm3=_mm_unpacklo_ps(xmm3,xmm4);
xmm8=_mm_shuffle_ps(xmm1,xmm3,_MM_SHUFFLE(3,2,3,2));

_mm_store_ps((float *)C + address, xmm8);
```

a)

```
xmm7=_mm_hadd_ps(xmm1, xmm1);
xmm7=_mm_hadd_ps(xmm7, xmm7);
_mm_store_ss((float *)C + address , xmm7);

xmm7=_mm_hadd_ps(xmm2, xmm2);
xmm7=_mm_hadd_ps(xmm7, xmm7);
_mm_store_ss((float *)C + address + 1 , xmm7);

xmm7=_mm_hadd_ps(xmm3, xmm3);
xmm7=_mm_hadd_ps(xmm7, xmm7);
_mm_store_ss((float *)C + address + 2 , xmm7);

xmm7=_mm_hadd_ps(xmm4, xmm4);
xmm7=_mm_hadd_ps(xmm7, xmm7);
_mm_store_ss((float *)C + address + 3 , xmm7);
```

b)

```
xmm9=_mm_unpacklo_ps(xmm1,xmm2);
xmm10=_mm_unpacklo_ps(xmm3,xmm4);
xmm7=_mm_shuffle_ps(xmm9,xmm10,_MM_SHUFFLE(1,0,1,0));
xmm8=_mm_shuffle_ps(xmm9,xmm10,_MM_SHUFFLE(3,2,3,2));
xmm8=xmm7+xmm8;

xmm9=_mm_unpackhi_ps(xmm1,xmm2);
xmm10=_mm_unpackhi_ps(xmm3,xmm4);
xmm7=_mm_shuffle_ps(xmm9,xmm10,_MM_SHUFFLE(1,0,1,0));
xmm10=_mm_shuffle_ps(xmm9,xmm10,_MM_SHUFFLE(3,2,3,2));
xmm10=xmm10+xmm7;
xmm8=xmm8+xmm10;

_mm_store_ps((float *)C + address, xmm8);
```

c)

Fig. 10 Three different ways for unpacking the multiplication results using SSE intrinsics; XMM1, XMM2, XMM3, XMM4 contain the C values. For most SIMD architectures, the three schedules are in increased performance order.

3.2 Multi-core CPUs

To run MMM effectively in many cores, the MMM problem is partitioned into smaller sub-problems and each sub-problem corresponds to a thread; each thread runs in one core only. Each thread must contain at least $p1$ instructions, where $p1$ is found experimentally and differs from one CPU to another. Otherwise, the cores will remain several CPU cycles idle since the threads initialization and synchronization time is made comparable to the threads execution time; this leads to low performance. Furthermore, in order to achieve high performance, the number of the threads must be higher than $p2$, where $p2$ is found experimentally. The impact of $p1$ and $p2$ on performance is comparable to the memory management problem.

Regarding small input sizes, a large speedup cannot be achieved. This is because either a low $p1$ and/or a low $p2$ value is selected. In this case it is preferable to run MMM in fewer number of cores in order to increase $p1$ and/or $p2$.

The MMM execution time on a quad core CPU is approximated by eq. 22 and eq. 23.

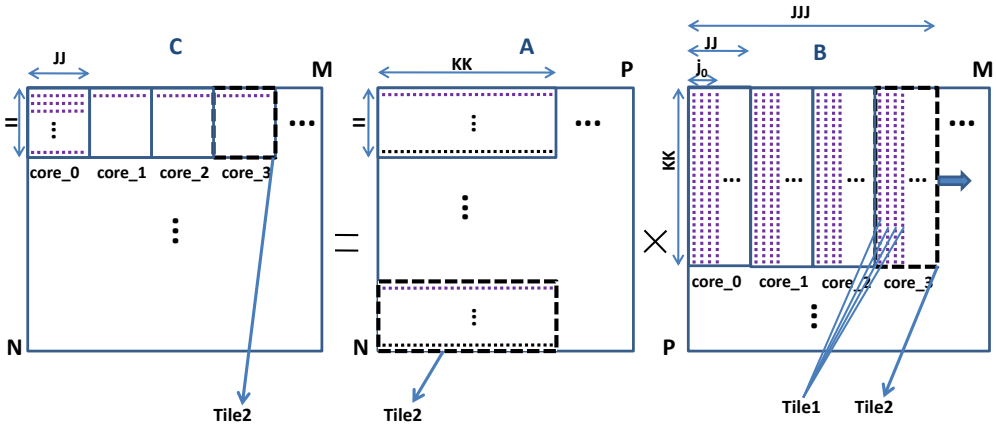

Fig. 11 The proposed methodology for 4 cores having a shared L2 cache.

$$T_{total} = max(T_{total-core1}, T_{total-core2}, T_{total-core3}, T_{total-core4}) \tag{22}$$

$$T_{total-corei} = \sum_{j=1}^{Num-of-Threads} T_{total-threadj} \tag{23}$$

where $i = [1, 4]$ and $T_{total-threadj}$ is the T_{total} value given in Subsect. 3.1.

Most multi core processors, typically contain 2 or 3 levels of cache, i.e., a) separate L1 data and instruction caches and a shared L2 cache, b) separate L1 data and instruction caches, separate unified L2 caches and a shared L3 cache. The proposed methodology for shared L2 and shared L3, is given in Subsect. 3.2.1 and Subsect. 3.2.2, respectively.

3.2.1 CPUs with shared L2

To utilize L2 shared cache, we partition the three arrays into Tile2 tiles (Fig. 11). The Tile2 tiles of A, B and C, are of size $II \times KK$, $KK \times JJ$ and $II \times JJ$, respectively. Each multiplication between two Tile2 tiles creates a different thread. For small and medium input sizes, we always select $KK = P$ (Fig. 11). Each multiplication between two Tile2 tiles is made as in Subsect. 3.1.5. Having q number of cores, each Tile2 of A is multiplied by q consecutive Tile2 tiles of B in parallel, each one at a different core (Fig. 11); thus, JJJ ($JJJ = q \times JJ$) is evenly divisible by M.

One Tile2 of A and at least q Tile1 of B have to fit in L2 shared cache. The Tile2 of A is always fetched to all the cores. Also, q Tile1 tiles of different Tile2 tiles of B are loaded, which have no consecutive elements between themselves. The goal is these q Tile1 tiles of B and the next four ones, do not conflict with the Tile2 of A and do not conflict with each other. In general, an L2 cache

with $assoc \geq q+1$ is needed here. Cache size equal to $((assoc-q) \times \frac{L2}{assoc})$ and $(q \times \frac{L2}{assoc})$ is needed for A (array A is written into main memory tile-wise) and B-C, respectively (ineq. 24). We select a big Tile2 of A to fit in L2 since having one Tile1 of B in each one of the q L1 data caches, their elements need to be multiplied by as many rows of A as possible before they are spilled to L2; Tile2 of A is reused M/j_0 times. L2 cache size that equals to q L2 ways is used for the Tile1 tiles of B and C, since their size is small and their elements are not reused (Tile1 of B is reused in L1 data cache not in L2).

Regarding L2 data cache, the largest II value which satisfy ineq. 24 is selected (Fig. 11). Given that a Tile1 of B is written in L1 data cache, it is memory efficient to be multiplied by as many rows of A as possible, before it is spilled from L1.

$$\frac{L2 \times (assoc - q)}{assoc} \geq II \times KK \tag{24}$$

where the KK value is determined according to L1 data cache size (ineq. 17). JJ value depends on the number of instructions each thread must contain and it is found experimentally; if JJ is smaller than this minimum number, thread initialization and synchronization time is comparable with its execution time. The large number of ways needed here is not a problem as the L2 associativity is larger or equal to 16 in most architectures.

The scheduling follows; suppose that there are four cores (Fig. 11). Each Tile2 of A is multiplied by a Tile2 of B exactly as in Subsect. 3.1.5. Each multiplication between two Tile2 tiles makes a different thread and each thread is executed at only one core. First, the first Tile2 of the first Tile2 block column of A is multiplied by all Tile2 tiles of the first Tile2 block row of B (M/JJ different threads); all M/JJ threads are executed in parallel exploiting the data reuse of Tile2 of A. Then, the second Tile2 of the first Tile2 block column of A is fetched and it is multiplied by all the Tile2 of the first Tile2 block row of B as above, etc. The procedure ends when all Tile2 block columns of A have been multiplied by all Tile2 block rows of B. Concerning main memory data accesses, A, B and C arrays are accessed 1, N/II and P/KK times, respectively.

3.2.2 CPUs with shared L3

To utilize L3 cache, A and B arrays are further partitioned into Tile3 tiles, of size $((q \times II) \times KK)$ and $(KK \times JJJ)$, respectively (Fig. 12). For small and medium input sizes, we always select $KK = P$ (Fig. 12). Each multiplication between two Tile2 tiles of A and B makes a different thread. Each multiplication between two Tile2 tiles is made as in Subsect. 3.1.5. We determine the Tile1, Tile2 and Tile3 parameters by the data cache sizes and associativities.

Regarding L2 cache, we compute the II value according to the L2 architecture parameters (ineq. 25).

$$\frac{L2 \times (assoc - 1)}{assoc} \geq II \times KK \tag{25}$$

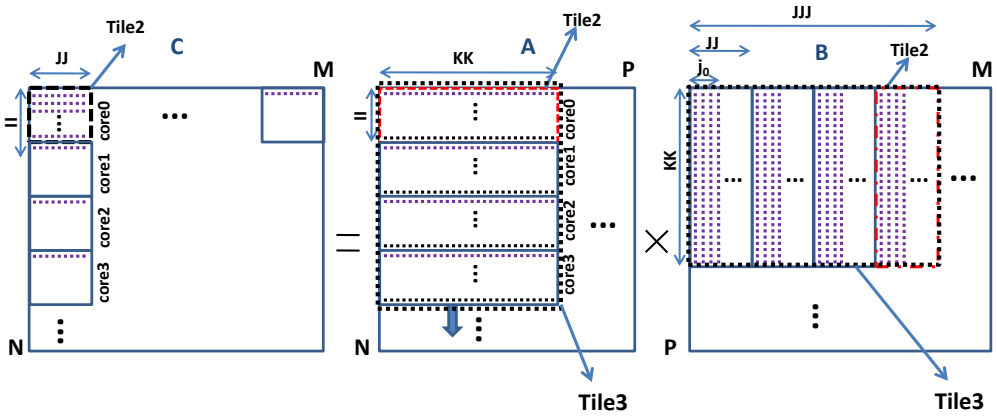

Fig. 12 The proposed methodology for 4 cores having a shared L3 cache.

The largest II value is selected satisfying ineq. 25. JJ is found experimentally since each thread has to contain a minimum number of instructions. KK is found according to the L1 data cache size (ineq. 17).

Given that a Tile1 of B is written in L1 data cache, it is memory efficient to be multiplied by as many rows of A as possible, before it is spilled from L1. Thus, $\frac{L2\times(assoc-1)}{assoc}$ size of L2 is used for A; the layout of A is tile-wise here. L2 cache size that equals to one L2 way is used for the Tile1 of B and C, since their size is small and their elements are not reused (Tile1 of B is reused in L1 data cache not in L2).

It is important to say that tiling for L2 is not always performance efficient because a) an L2 miss has a small penalty here; this is because in this case, the data are loaded not from main memory but from L3 cache, which is fast enough, b) extra addressing instructions are inserted which may degrade performance. Thus, tiling only for L1 and L3, may be more efficient in several cases.

Regarding L3 cache, we compute JJJ according to the L3 cache parameters. We choose the biggest Tile3 of B possible, to fit in L3 shared cache. There is $((assoc-k-1)\times\frac{L3}{assoc})$ L3 size for the Tile3 of B, $(k\times\frac{L3}{assoc})$ for A and $(\frac{L3}{assoc})$ for C (ineq. 26).

$$((assoc-k-1)\times\frac{L3}{assoc})\geq KK\times JJJ \tag{26}$$

where $k=\lceil\frac{q\times II\times KK}{L3/assoc}\rceil$ and L3 is the size of the L3 cache. The larger JJJ value is selected satisfying ineq. 26. Also, $JJJ=l\times JJ$ where l is an integer. JJ value depends on the number of instructions, each thread must contain and it is found experimentally. KK value is found according to L1 data cache size and it is given by ineq. 17.

The elements of Tile3 of A and B are written in consecutive memory locations in main memory and thus they occupy consecutive cache lines in L3 cache. These two Tile3 tiles must use different L3 cache ways as cache lines

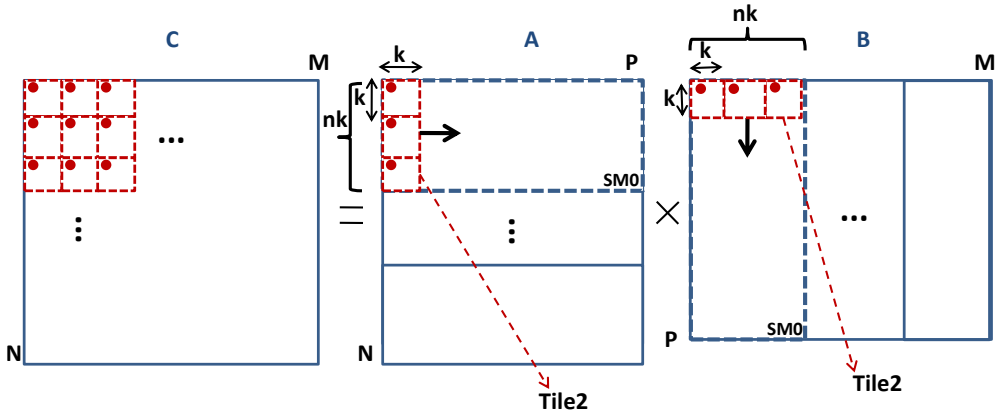

Fig. 13 MMM for GPU. The bullets with the red color show the elements accessed/multiplied by Thread0 only.

of Tile3 of B must not conflict with Tile3 of A ones. Cache size equals to one L3 way is used for C array for its cache lines not to conflict with the Tile3 of A and B ones; C elements are not reused and they are not occupying a large space.

The scheduling follows; suppose that there are 4 cores (Fig. 12). Each Tile2 of A is multiplied by a Tile2 of B exactly as in Subsect. 3.1.5. Each multiplication between two Tile2 tiles makes a different thread and each thread is executed in one core only. First, all Tile2 tiles of the first Tile3 of A are multiplied by all Tile2 tiles of the first Tile3 of B ($(JJJ/JJ) \times q$ different threads); these threads are executed in parallel exploiting the data reuse of Tile1 of B in L1, the data reuse of Tile2 of A in L2 and the data reuse of Tile3 of B in L3. Then, the same Tile3 of B as above, is multiplied by all the Tile2 tiles of the second Tile3 of the first Tile3 block column of A, etc; each Tile3 of B is reused N/II times (this is why the Tile3 of B has to fit in L3 shared cache) and each Tile2 of A is reused (JJJ/j_0) times in L2 (this is why Tile2 of A has to fit in L2 cache). The procedure is repeated until all Tile3 block columns of A have been multiplied by all Tile3 block rows of B.

3.3 GPUs

All modern GPU architectures are in common (compute capability of 2.∗, 3.∗). They consist of p (p is up to 16) Streaming Multiprocessors (SM) which are connected to a common L2 cache. Each SM contains k processors (k is up to 192), each one having a configurable L1; L1 memory contains an L1 data cache and a shared L1. The number of the SMs, the number of the processors and the memories sizes differ from one architecture to another. Also, GPUs have smaller and faster cache memories in contrast to the CPUs.

Regarding MMM performance on GPUs, the most critical parameters are the number of the threads run in parallel, the number of the SMs work in parallel, the GPU occupancy and the memory management.

Table 2 Number of data accesses in memory hierarchy

	C	A	B
shared L1	0	$r \times \frac{n \times k^2 \times P}{cores_{SM}}$	$r \times n \times P$
L1 data cache	$\frac{n^2 \times k^2}{cores_{SM}}$	0	0
L2	$N \times M$	$\frac{N \times P \times M}{n \times k}$	$\frac{P \times M \times N}{n \times k}$
DDR	$N \times M$	$\frac{N \times P \times M}{n \times k}$	$\frac{P \times M \times N}{n \times k}$

The proposed methodology gives a different schedule according to the GPU architecture parameters and to the input size.

Likewise Subsect. 3.1.1, the best schedule here is that of row-column for two reasons. First, this schedule gives the minimum number of DDR accesses. This is because C array is not just loaded, but it is also stored into main memory (it is accessed twice); thus it is preferable to access C array only once and A, B more times than the opposite (Subsect. 3.1.1). Second, by writing the C array just once to main memory, we avoid synchronization problems and all threads run in parallel.

To utilize the lower level GPU memories, the three arrays are partitioned into smaller ones according to the number of the registers and the number of the threads, each SM supports. The three arrays are partitioned into square tiles of size $k \times k$, where k is a power of 2 and $k^2 \leq Num\_Threads$, where $Num\_Threads$ is the number of the threads each SM supports. Furthermore, $n1$ Tile1 tiles of A and $n2$ Tile1 tiles of B constitute a Tile2 of A and B, respectively (Fig. 13); we select $n = n1 = n2$ and $n \geq 2$ (a detailed analysis is given below). n depends on the input size and its maximum value is limited to the number of the available registers. A different n value is selected for different input sizes in order to achieve high occupancy (it is explained below).

The number of blocks of threads run in parallel is $\frac{N}{k \times n} \times \frac{M}{k \times n}$. In order to none of the SMs remains idle, ineq. 27 holds. Ineq. 27 satisfies that the number of the blocks of threads is always larger or equal to the number of the SMs; otherwise, several SMs will remain idle.

$$\frac{N \times M}{k^2 \times n^2} \geq Num\_SMs \tag{27}$$

We select the largest k value possible satisfying ineq. 27, where $n \geq 2$. We select the largest k value as by increasing k, a) the number of the threads run in parallel increases, b) the number of data accesses is decreased; in modern GPU architectures, always $k \geq 16$.

The schedule follows. There are $\frac{N}{k \times n} \times \frac{M}{k \times n}$ blocks of threads and each block contains k^2 threads. The bullets shown with the red color in Fig. 13, are multiplied by each other, during the execution of thread0; in this case, $n = 3$

and thus there are 9 intermediate results of C (the intermediate results are produced and consumed since they are in registers). The thread1 multiplies the 3 elements of A shown in Fig. 13 by the 3 elements of B that exist by one position to the right of that shown in Fig. 13. The thread2 multiplies the 3 elements of A shown in Fig. 13 by the 3 elements of B that exist by two positions to the right of that shown in Fig. 13 etc. The thread of number k, multiplies the 3 elements of A that exist one position down to those shown in Fig. 13 by the 3 elements of B shown in Fig. 13 etc. In general, each thread multiplies n elements of the first column of Tile2 of A by n elements of the first row of Tile2 of B (k^2 threads are executed in parallel). Then the procedure continues with the second column of Tile2 of A and the second row of Tile2 of B etc. The multiplication of the first $n \times k$ rows of A by the first $n \times k$ columns of B takes place in the first SM. The multiplication of the first $n \times k$ rows of A by the second $n \times k$ columns of B takes place in the second SM etc.

The number of data accesses in memory hierarchy is given in Table 3, where $r = (\frac{N}{k \times n} \times \frac{M}{k \times n})/(Number\_SMs)$ and $cores_{Sm}$ is the number of the cores in each SM (in Table 3, we assume that none of the cores remains idle). In Table 3, the Shared L1 and the L1 data cache value corresponds to each core.

Tile2 tiles of A and B achieve data reuse and thus they are placed in Shared L1 (ineq. 28).

$$Shared\_L1 \geq 2 \times n \times k^2 \tag{28}$$

We use shared L1 and not L1 data cache. An implementation with Shared L1 is faster than L1 data cache because a) shared memory normally has 32 banks and is much less susceptible to conflicts, b) by using shared memory, the layout of A and B is not changed (except from special case explained below), decreasing the number of load/store and addressing instructions.

Shared L1 is fully utilized. Shared L1 access time equals to 1 clock if no data conflicts occur. Normally, shared L1 memory is organized into 32 banks and each bank has width of 32 bits; successive 32-bit words are assigned to successive banks. If all threads of a warp access different banks, there is no bank conflict. For most GPU architectures, if $k \geq 16$ no bank conflicts exist. On the other hand, if $k \prec 16$, the data array layouts are changed from row-wise to tile-wise in order to eliminate L1 conflicts, i.e., first we write the first tile's elements in main memory in the exact order they are loaded, then the second's etc.

Another critical parameter for GPU is memory coalescing. Normally, DDR memory accesses data in 64/128 byte segments. Thus, to achieve peak bandwidth transfer rate, we have to access data within 64/128 byte boundaries at the minimum. Regarding array A, this means that in the case that $k \prec 32$ and DDR memory accesses data in 128 byte segments, we have to change the A data array layout from row-wise to tile-wise (we suppose that the arrays are floating point numbers). Regarding array B, this means that in the case that $n \times k \neq 32 \times m$, where m is a positive integer, the data array layout of B is changed from row-wise to tile-wise.

To sum up, whether the data arrays layouts change or not, depends on the shared L1 / DDR memory architecture parameters and on the input size. However, the change of the data arrays layouts introduces an additional cost. The arrays have to be loaded and rewritten from/to DDR memory. To find out whether changing the data arrays layouts is performance efficient or not, the two schedules are tested and the fastest is selected. Normally, if the above parameters are very close to the optimum ones, performance is approximately the same.

Regarding L2 data cache, tiling is not performance efficient; this is because L2 size is small in contrast to the number of the SMs; if we partition the three arrays even more into Tile3 tiles in order to the new tiles fit in L2, the tiles sizes would be very small and the extra addressing and load/store instructions will overlap the locality advantage.

The MMM execution time can be approximated by eq.1. Moreover, if we assume that none of the cores remains idle $T_{matrix-operations}$ (eq.3) is transformed into eq. 29; the floating point multiplications and additions are implemented by the multiply-add instructions which all the GPUs support.

$$T_{matrix-operations} = \frac{Multiply - add - latency \times (N \times M \times P)}{Number - of - cores} \tag{29}$$

Also, T_{data} is approximated by the following equation.

$$\begin{aligned} T_{data} = max(& \frac{L1_{load\_lat} \times L1_{loads}}{load/store\_Units}, L1_{Store\_lat} \times \frac{L1_{Stores}}{L1_{ports}}, \\ & \frac{L2_{load\_lat} \times L2_{loads}}{line_{Shared}} + \frac{L2_{store\_lat} \times L2_{stores}}{line_{L1}}, \\ & \frac{DDR_{load\_lat} \times DDR_{loads}}{line_{L2}} + \frac{DDR_{store\_lat} \times DDR_{stores}}{line_{L2}}) \end{aligned} \tag{30}$$

If we assume that shared L1, L2 and DDR have access latencies of 1 cycle, 3 and 50 cycles, respectively and that $line_{L1} = line_{L2} = 4$, $L1_{ports} = 2$, $line_{Shared} = 4$, $cores_{SM} = 32$ and $load/store\_Units = 16$, eq. 30 and Table 3 give:

$$\begin{aligned} T_{data} = max(& r \times \frac{n \times k^2 \times P}{32} + r \times n \times P, \frac{n^2 \times k^2}{32}, \\ & \frac{3 \times (N \times M + \frac{2 \times N \times P \times M}{n \times k})}{4} + \frac{3 \times N \times M}{4}, \\ & \frac{50 \times (N \times M + \frac{2 \times N \times P \times M}{n \times k})}{4} + \frac{50 \times N \times M}{4}) \end{aligned} \tag{31}$$

The number of DDR accesses is the critical parameter. If $n1 \neq n2$ had been used instead of $n = n1 = n2$, then eq. 31 would had $(\frac{N \times P \times M}{k} \times (\frac{1}{n1} + \frac{1}{n2}))$ instead of $(\frac{2 \times N \times P \times M}{n \times k})$. This is because array A is loaded $M/(n2 \times k)$ times,

B is loaded $N/(n1 \times k)$ and C is loaded/stored once. Thus, the total number of loads is $\frac{N \times M \times P}{k} \times (\frac{1}{n1} + \frac{1}{n2}) + N \times M$. $(\frac{1}{n1} + \frac{1}{n2})$ value is minimized for $n1 = n2$. This is why we use $n1$ Tile1 tiles of A and $n2$ Tile1 tiles of B, where $n = n1 = n2$.

According to eq. 31, the DDR/L2 access time is minimized when $n \times k$ is maximized. However, k, n maximum values are $k \leq 32$ and $n \leq 6$, respectively, because their values are restricted to the number of the registers and threads, each SM supports.

The best T_{data} value does not necessarily gives the best performance. If the numbers of the a) threads run in parallel and/or b) SMs work in parallel, are less than the number of the cores and the number of the SMs, respectively, performance is degraded.

Regarding small input sizes, we select a small n value according to ineq. 27 ($n = 2$), because if we don't, the number of blocks of threads will become smaller than the number of the SMs; this means that several SMs will remain idle, decreasing performance. Since the number of blocks of threads is $\frac{N}{k \times n} \times \frac{M}{k \times n}$, their number is increased when n is decreased.

Regarding medium input sizes, n value is up to 4. For large input sizes, $n \succ 4$. Performance is increased by increasing the n value since a smaller number of data accesses is achieved.

MMM performance is also increased by applying software prefetching by using software pipeline. In this case, we use two times more shared L1 memory than ineq. 28, since we need the current Tile2 tiles (Tile2 tiles of A and B) and the next ones. When the current Tile2 tiles are multiplied by each other, the next Tile2 tiles are loaded from DDR to shared L1, in parallel; in this way, the cores do not remain idle until the Tile2 tiles are fetched.

Finally, performance is slightly increased by applying loop unroll transformation and by using the texture memory to store the A and B arrays on devices of compute capability larger than 2.0. By using texture memory, a) it does not interfere with the other caches (this is important if there is a lot of pressure on the L1/L2 caches) b) it supports address computations.

4 Experimental Results

The experimental results for the proposed methodology, presented in this section, were carried out with Intel Xeon CPU E3-1241 v3 (four physical cores), Pentium Intel i7-2600K (6 physical cores), Valgrind tool [28], ARMv7-a on GEM5 simulator [29], PowerPC-440 on Xilinx FPGA Virtex-5, GEM5 and SimpleScalar simulator [30] and Nvidia eForce GGTX-580 of computate capability 2.0. Optimization level -O3 was used at all cases. The three arrays are one dimensional arrays and their elements are aligned into main memory according to the L1 data cache line size, since the aligned load/store instructions have lower latency than the no aligned ones. The routine changing the arrays layouts is always included to the execution times.

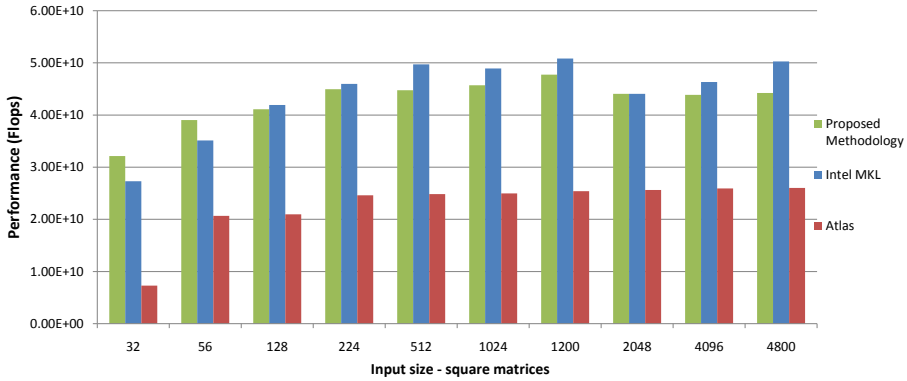

Fig. 14 Performance evaluation over ATLAS and Intel_MKL state of the art software libraries on the one core of Intel Xeon CPU E3-1241 v3 for square matrices

Although a performance comparison with Intel_MKL is unfair, a detailed experimental analysis has been made as it is the fastest MMM library in the world for Intel general purpose processors. A performance comparison with Intel_MKL is unfair for two reasons. First, Intel_MKL developers have access to all the Intel processor architecture details which we do not, e.g., victim cache, hardware prefetchers; this is why Intel_MKL library is the fastest library on Intel processors only. Second, Intel_MKL loop kernels are written in assembly code while our method in C (assembly code is always more efficient); Intel developers write assembly code to deal with the low level transformations, e.g., register allocation, instruction selection and instruction scheduling. The proposed methodology lies at a higher level of abstraction and it is used to wide range of computer architectures. Implementing the proposed methodology in assembly code is beyond the scope of this paper and thus the low level transformations are applied by the target compiler (which is less efficient). The scope of this paper is not to provide the peak-performance MMM implementations, but to analytical give the architecture dependent high level transformation parameters (e.g., tile sizes) that achieve peak-performance. We strongly believe that if could modify the MKL library scheduling parameters according to the proposed methodology, an even higher performance would be achieved.

The experimental results for single-core and multi-core CPU architectures are given in Subsect. 4.1 and Subsect. 4.2, respectively. The experimental results for GPU architectures are given in Subsect. 4.3. Finally, in Subsect. 4.4, an evaluation between multi-core CPUs and GPUs is given.

4.1 Experimental Results for single-core CPU architectures

Regarding general purpose processors, a performance comparison is made over ATLAS SOA library 3.10.2 and Intel_MKL (Parallel Studio XE 2016) by using the one of the four cores of Intel Xeon CPU E3-1241 v3. The Intel processor

Table 3 Number of load/store instructions and DDR data accesses of Intel_MKL, ATLAS and the proposed methodology, respectively, of the Fig. 14 case (Valgrind tool is used)

Size	Intel_MKL		ATLAS		Proposed Methodology	
	L/S instr.	DDR data acc.	L/S instr.	DDR data acc.	L/S instr.	DDR data acc.
56	$5.22E+06$	$1.14E+05$	$3.36E+06$	$9.37E+03$	$3.43E+06$	$8.72E+03$
224	$5.88E+07$	$9.47E+04$	$5.63E+07$	$2.92E+04$	$5.78E+07$	$2.79E+04$
512	$3.11E+08$	$1.83E+05$	$3.14E+08$	$1.58E+05$	$3.14E+08$	$1.45E+05$
1024	$1.39E+09$	$2.06E+06$	$1.43E+09$	$3.45E+06$	$1.40E+09$	$1.89E+06$
2048	$6.18E+09$	$1.20E+07$	$6.49E+09$	$2.36E+07$	$6.22E+09$	$1.15E+07$

contains four physical cores (each core contains 16 YMM registers of 256-bit each), each one with 32 kbyte L1 data and instruction caches (8-way associative) and 256 kbyte L2 unified cache. The cores use a shared L3 cache of size 8 Mbytes. The Operating system Ubuntu 14.04 and the gcc-4.8.4 compiler are used. The cache latency values in CPU clocks for $E3-1240$ are $L1_{load\_lat}=4$, $L2_{load\_lat}=11$ and $L3_{load\_lat}=42$. The Operating system Ubuntu 14.04 and the gcc-4.8.4 compiler are used. It is important to say that by using only the one core, the one thread has to be manually assigned to the one core; the programmer has to set the CPU thread affinity flag. Otherwise, the operating system (OS) will make the core assignment, and it will toggle the thread among the cores degrading performance because of the pure data locality. The proposed methodology is compared with *cblas_dgemm* library (double precision values) for square and non-square matrix sizes (Fig. 14, Fig. 15). Also, the Valgrind tool [28] is used to measure the number of load/store instructions and the number DDR memory accesses of the three methods for Fig. 14. In all the figures shown in this section the average execution time among 10 executions is shown.

First, a performance evaluation for square matrices is given. Regarding small input sizes (we used the schedules given in Subsect. 3.1.1 and Subsect.3.1.6), i.e., $N=32$ and $N=56$ (Fig. 14), the proposed methodology achieves a very large performance gain over ATLAS and a significant gain over Intel_MKL. The proposed methodology achieves the smallest number of DDR accesses (Table 3); moreover, it achieves a much smaller number of load/store instructions than MKL and about the same number with ATLAS (Table 3). The number of load/store instructions is less because the number of YMM registers has been fully exploited; 14, 1 and 1 YMM registers are used for C, A and B arrays, respectively ($p=14$ in Fig.9). For $N=56$ case, both the proposed methodology and ATLAS execute about 8.5×10^6 instructions in total (both arithmetical and load/store instructions) while Intel_MKL executes 13.9×10^6. We believe that the above number of arithmetical instructions here is close to the minimum because we apply loop tiling transformation to the one of the three iterators only (j iterator), decreasing the number of addressing instructions. Although ATLAS achieves a lower number of load/store instructions, arithmetical instructions and DDR accesses than Intel_MKL, it does achieve performance gain; we strongly believe that the reason is the bet-

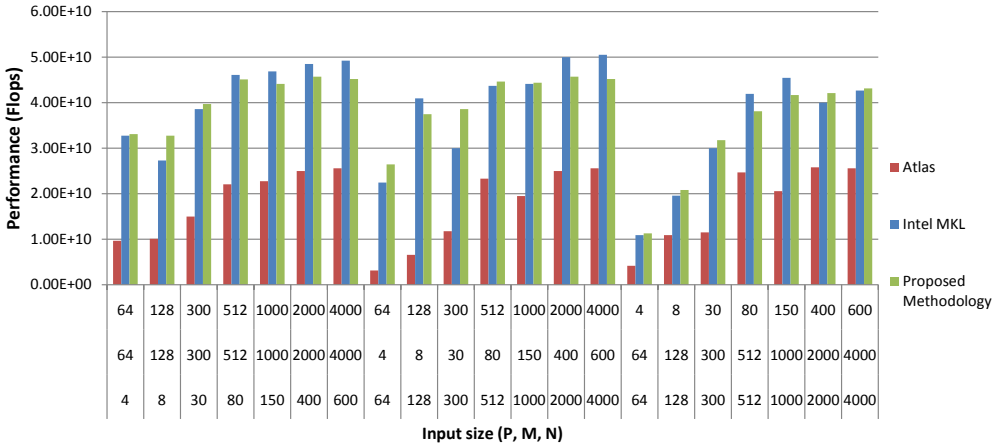

Fig. 15 Performance evaluation over ATLAS and Intel_MKL state of the art software libraries on the one core of Intel Xeon CPU E3-1241 v3 for non-square matrices

ter Intel_MKL hand-written assembly code and the more efficient low level optimizations it applies (they are further explained below).

For medium and large input sizes (we used the schedules given in Subsect.3.1.4 and Subsect.3.1.5), the proposed methodology achieves a large performance gain over ATLAS (speedup value is about 1.8) and a very small performance loss over Intel_MKL (from 1.01 up to 1.13 times slower). Regarding the number of load/store instructions, the three methods achieve approximately the same values. As far as the number of DDR data accesses is concerned, the proposed methodology achieves a smaller number at all cases. This is because for different input size and cache parameters a different schedule is generated minimizing the number of data accesses; this is also shown in Table 1, where the numbers of memories accesses are shown given the tile sizes.

Given that the proposed methodology achieves about the same number of load/store and arithmetical instructions and a lower number of DDR accesses (for medium and large input sizes it the most performance critical parameter) than MKL at all cases, we strongly believe that the small performance loss is due to the four following low level optimizations: a) MKL uses a more efficient AVX instruction set (especially when unpacking and storing the multiplication results), b) MKL achieves a lower Taddressing value due to the more efficient hand-written assembly code, c) the sequence of MKL assembly instructions is more efficient and thus the number of idle cycles in the pipeline is minimized, d) MKL uses in a more efficient way some Intel processor units (e.g., victim cache, hardware prefetchers) since it has access to the hardware details, e) software prefetch instructions are written at the exact code line. We strongly believe that if could modify the MKL library to use the proposed methodology, a higher performance would be achieved.

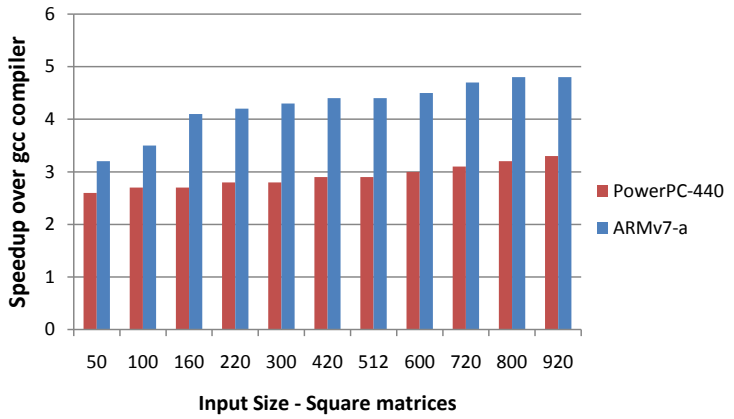

Fig. 16 Performance evaluation over PowerPC-440 on Xilinx Virtex-5 FPGA and ARMv7-a on GEM5 simulator

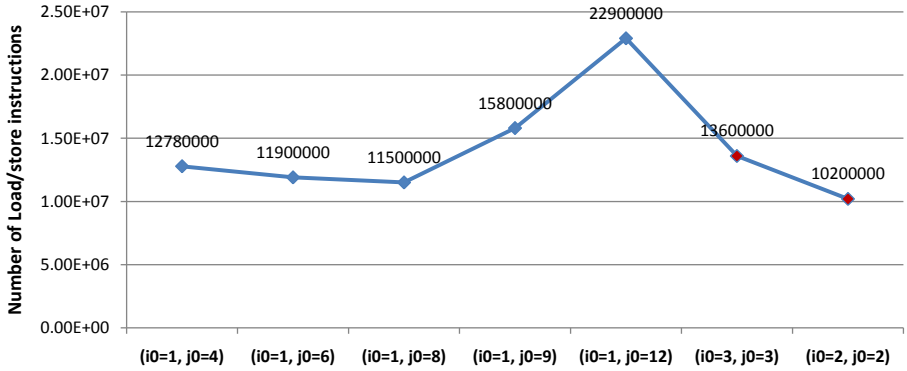

Fig. 17 Evaluation of Subsect. 3.1.1 - Number of load/store instructions for different tile sizes - ineq.5 holds.

As far as non-square matrices are concerned, the performance is about the same as in Fig. 14 except from the third cluster of results (P dimension is much smaller than the others). In this case, Intel_MKL performance is reduced and the peak performance of 50 Gigaflops is not achieved. ATLAS does not perform well for small input sizes but for medium and large input sizes achieves about 25 Gigaflops at all cases. The proposed methodology is the fastest for small input sizes and also it is comparable to Intel_MKL for medium and large input sizes. In general, the proposed methodology and ATLAS achieve about the same number of flops for square and non-square input sizes, while MKL does not.

The proposed methodology is also compared with two different embedded processors, i.e., PowerPC-440 on Xilinx Virtex-5 FPGA and ARMv7-a on GEM5 simulator (Fig. 16). Given that there is no state of the art software library for these processors, we evaluated our methodology with 'arm-linux-gnueabihf-gcc' and Xilinx compiler, on ARM and PowerPC, respectively. As

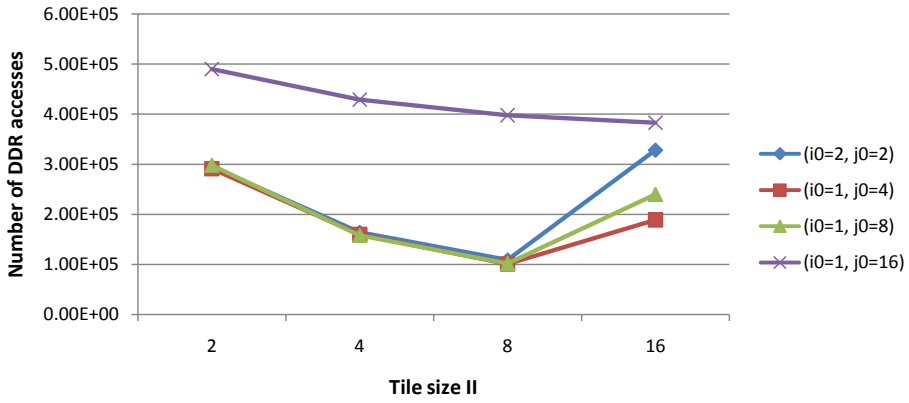

Fig. 18 Evaluation of Subsect. 3.1.2 - Number of DDR accesses for different tile sizes - ineq. 9 holds (Fig. 5).

far as PowerPC 440 is concerned, it is a 32-bit embedded Superscalar processor with 7 stage pipeline, developed by IBM. It contains L1 data and L1 instruction cache memories of size 32 kbytes, highly-associative (64-way). As far as ARMv7-a is concerned, it is a RISC dual core processor with two levels of data cache (experiments are made on the one of the two cores). It contains L1 data and instruction caches of 32kbytes (2-way associative) and L2 cache of 1Mbyte and 16-way associative. The speedup values are from 2.6 up to 3.3 and from 3.2 up to 4.8, for PowerPC and ARM, respectively. PowerPC performs better than ARM because the data cache of the first processor is 64-way associative while the data cache of the second is 2-way associative; this gives a much larger number of cache misses. As the input size increases, the speedup value increases (at both processors) as the memory management problem becomes more critical.

Furthermore, an evaluation of Subsect. 3.1.1 (small input sizes - ineq.5 holds) is made on SimpleScalar simulator (Fig. 17). In Fig. 17, the number of load/store instructions is shown, for different tile sizes; square matrices of size $N = 216$ are taken. Since ineq.5 holds, all the data of B and $2 \times i_0$ rows of A, fit in L1 data cache (Fig.2). In SimpleScalar simulator, 10 available floating point registers exist and according to ineq.6, the best schedule regarding the minimum number of load/store instructions is $(i_0 = 2, j_0 = 2)$ and the next is $(i_0 = 1, j_0 = 8)$ (the first uses 8 registers and the second 10 registers); the first schedule achieves a slightly smaller number of load/store instructions since the following equation $2 \times N \times M + \frac{N \times P}{i_0} + \frac{P \times M}{j_0}$ (it gives the number of load/store instructions) gives $2 \times N^2 + N^2 = 3 \times N^2 \prec 2 \times N^2 + N^2 + N^2/8 = 3 \times N^2 + N^2/8$. Furthermore, by using more than 10 registers, the number of load/store instructions is highly increased since the data are spilled from the RF and reloaded many times (Fig. 17). The analyss made in Subsect. 3.1.1 is confirmed by Fig. 17.

Moreover, an evaluation of Subsect. 3.1.2 is made on on SimpleScalar simulator (Fig. 18). One level of cache is selected here with L1 data cache of size

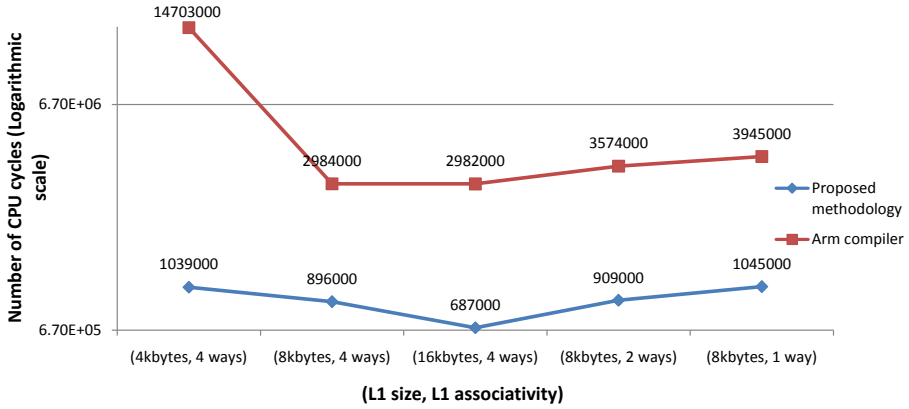

Fig. 19 Performance evaluation over different cache size and associativity values

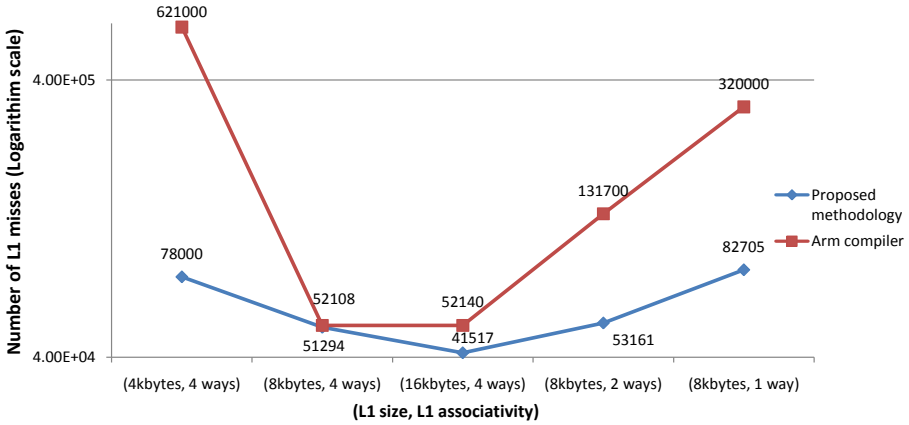

Fig. 20 Evaluation over different cache size and associativity values

8 Kbytes and 8-way associative (each way is of size 1 kbyte); given that there is no L2 cache, the number of L1 misses equals to the number of the DDR accesses. The number of DDR accesses / L1 misses for different tile sizes is shown, when ineq. 9 holds (Fig. 18) (floating point elements and $N = M = P = 128$). As it was expected, as far as ineq. 9 holds, i.e., if the II rows of A and the j_0 columns of B fit in L1, the number of DDR accesses is kept low (the B array is written tile-wise in main memory). For the $(i_0 = 1, j_0 = 16)$ case, ineq. 9 does not hold for none $II \succ 1$ value, as the Tile1 of B is 8 kbytes; this is why a large number of DDR accesses is achieved at all different II values. Regarding, $(i_0 = 1, j_0 = 4)$, $(i_0 = 1, j_0 = 8)$ and $(i_0 = 2, j_0 = 2)$ cases, the maximum II value satisfying ineq. 9 is $II = 8$; this is why the minimum number of DDR accesses is achieved for this case. When $II = 16$, Tile1 of A becomes 8 kbytes and thus there is no cache space for Tile1 of B, increasing the number of DDR accesses.

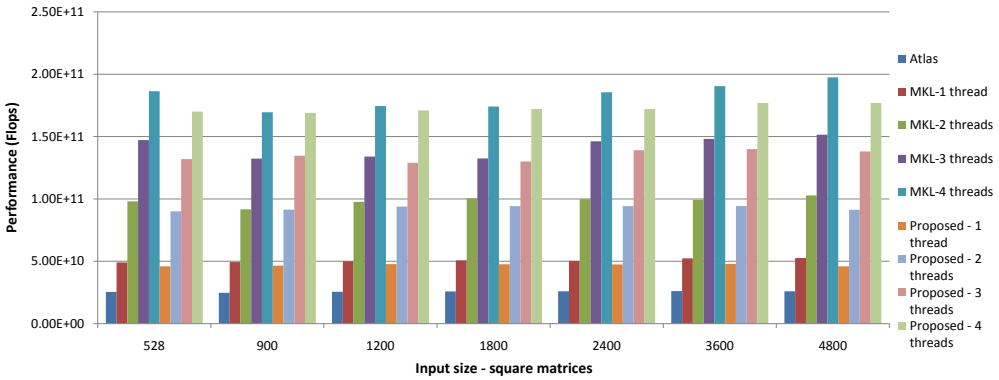

Fig. 21 Performance evaluation on Intel Xeon CPU E3-1241 v3 (4 physical cores exist), by using more than one physical cores

Finally, an evaluation over different cache size and associativity values is made on GEM5 simulator using ARMv7-a processor for five different cache sets (L1 size, L1 associativity) (Fig. 19 and Fig. 20). The compiler used is 'arm-linux-gnueabihf-gcc' and the input size is $N = 80$ (floating point values and square matrix sizes). Regarindg gcc compiler, by changing the cache parameters, its performance does not change significantly, except from the case that L1=4 kbytes, where its performance is about 5 times lower; this is because the row of A cannot remain in L1 cache in this case and both A and B elements are always loaded from DDR. In order to the row of A remains in L1, one row of A and one column of B have to be smaller than L1 size (otherwise, the A elements are spilled); however, B is written row-wise in main memory (by default) and all the column elements are written in no consecutive main memory locations; thus for one column element to be fetched, an entire L1 cache line is fetched (cache line size is 64 bytes). Given that each array element is 4 bytes (float numbers), for one column of B to be fetched, 5120 bytes are necessarily fetched (16 clumns of B which are of size $80 \times 16 \times 4$), and this is why performance is highly decreased when L1=4 kbytes. This is also shown in Fig. 20 where the number of L1 misses is highly increased. Another important observation is that in contrast to gcc, the proposed methodology achieves a much smaller number of L1 misses for small associativity values (Fig. 20); this is because the data cache associativity is taken into account. Moreover, by increasing the cache size, the proposed methodology further increases its performance, while gcc does not.

4.2 Experimental Results for multi-core CPU architectures

An evaluation of the proposed methodology by using more than one physical cores is made in Fig. 21. The experimental results were carried out with Intel Xeon CPU E3-1241 v3 by using the AVX instructions. The proposed methodology is compared with ATLAS SOA library 3.10.2 and Intel_MKL (Parallel

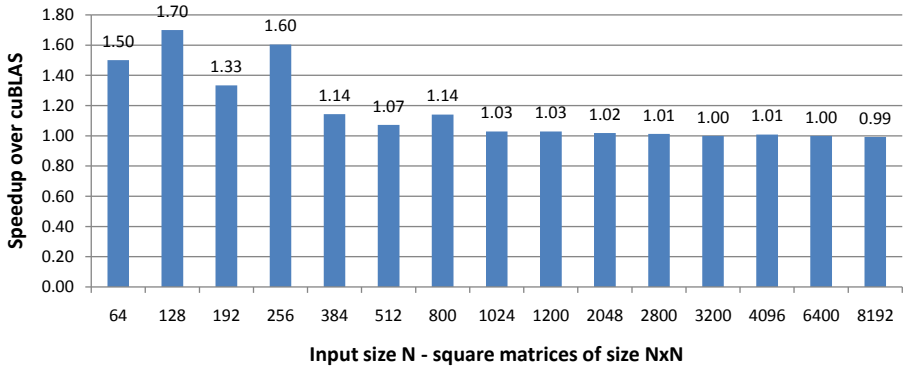

Fig. 22 Speedup over cuBLAS state of the art software library on GPU GTX580

Studio XE 2016) (cblas_dgemm routine). It is important to say that ATLAS does not support multiple threads. As far as the performance on the one core is concerned, the results are similar to Fig. 14. By increasing the number of the threads, performance is increased accordingly. The CPU utilization factors are from 1.85 up to 1.95, from 2.67 up to 2.9 and from 3.4 up to 3.79, for 2,3 and 4 cores, respectively. Regarding small input sizes, the core utilization factor is smaller, as the thread initialization time is comparable to its execution time. The proposed methodology uses the schedule given in Subsect. 3.2.2. Performance is highly affected by the number of YMM registers used and by the L1 tile size. Regarding shared cache, performance is not highly affected by using different tile sizes, suffice the tiles fit in shared cache. Intel MKL achieves a small speedup over the proposed methodology for the reasons explained in the previous Subsection.

4.3 Experimental Results for GPU architectures

The experimental results presented in this Subsection, were carried out with Nvidia GeForce GTX-580. GTX-580 contains 16 SMs which are connected to a common L2 of size 768 Kbytes. Each SM contains 32 cores each one with its own configurable L1. The comparison is made with cuBLAS (CUDA Toolkit 6.5) state of the art library.

The proposed methodology achieves a significant speedup over cuBLAS SOA library for small and medium input sizes and approximately the same performance for large input sizes (Fig. 22). For input size $N = 64$ and $N = 128$, $n = 2$ and $k = 32$ are selected according to the corresponding inequalities given in Subsect. 3.3; if we use a larger n value for these cases, the number of blocks of threads will become smaller than the number of the SMs and performance will be degraded; the n value is selected according to the ineq. 27. For larger n sizes, we cannot select $k = 32$, since we exceed the available number of registers; thus, $k = 16$ is selected. Furthermore, for input sizes $N = 192$ and

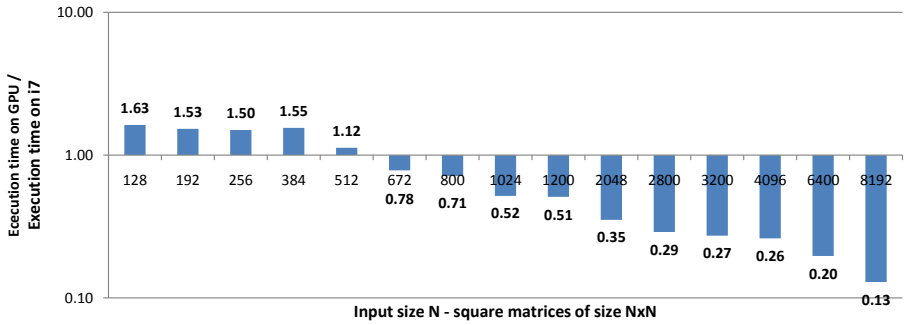

Fig. 23 Performance comparison between GPU GTX580 and CPU i7-2600K

$256 \leq N \prec 2048$, $n = 3$ and $n = 4$ are selected, respectively. Last, for input sizes $N \geq 2048$, $n = 5$ is selected.

4.4 Evaluation between multi-core CPUs and GPUs

Finally, an evaluation between GPU GTX580 and CPU i7-2600K (6 physical cores) is made in Fig. 23. The GPU contains 512 small cores while the CPU contains 6 big cores each one using SSE vector instructions; regarding floating point data, this means that each one of the CPU cores executes 4 floating point operations at each cycle. For small and medium sizes, i.e., $N \leq 512$, the CPU performs faster than GPU (Fig. 23); this is because in this case, the time needed to transfer the 3 arrays from the CPU to GPU and vice versa, is comparable to the execution time. However, for larger input sizes, where the time needed to transfer the 3 arrays from the CPU to GPU and vice versa, is smaller than the execution time, GPU performs faster. For $N = 1024$, $N = 2048$ and $N = 4096$, GPU is 1.92, 2.83 and 3.8 times faster, respectively. As the input size increases, the speedup over CPU increases, since the transfer time becomes smaller than the execution time.

5 Conclusions

In this paper, an MMM speeding up methodology is presented where the optimum scheduling parameters are found by decreasing the search space theoretically while the major scheduling sub-problems are addressed together as one problem and not separately. For the first time an MMM methodology is given for such a wide range of computer architectures. For different hardware architecture parameters, a different implementation is produced. This is achieved by fully exploiting the MMM algorithm characteristics and the hardware architecture parameters.

References

1. V. I. Kelefouras, A. Kritikakou, E. Papadima, C. E. Goutis, A methodology for speeding up matrix vector multiplication for single/multi-core architectures., The Journal of Supercomputing 71 (7) (2015) 2644–2667.
2. S. S. Pinter, Register allocation with instruction scheduling: a new approach, J. Prog. Lang. 4 (1) (1996) 21–38.
3. G. Shobaki, M. Shawabkeh, N. E. A. Rmaileh, Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach, ACM Trans. Archit. Code Optim. 10 (3) (2008) 14:1–14:31. doi:10.1145/2512432.
   URL http://doi.acm.org/10.1145/2512432
4. D. F. Bacon, S. L. Graham, Oliver, J. Sharp, Compiler transformations for high-performance computing, ACM Computing Surveys 26 (1994) 345–420.
5. E. Granston, A. Holler, Automatic recommendation of compiler options, in: In Proceedings of the Workshop on Feedback-Directed and Dynamic Optimization (FDDO, 2001.
6. S. Triantafyllis, M. Vachharajani, N. Vachharajani, D. I. August, Compiler optimization-space exploration, in: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, CGO '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 204–215.
   URL http://dl.acm.org/citation.cfm?id=776261.776284
7. K. D. Cooper, D. Subramanian, L. Torczon, Adaptive optimizing compilers for the 21st century, Journal of Supercomputing 23 (2001) 2002.
8. T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, F. Bodin, H. A. G. Wijshoff, A feasibility study in iterative compilation, in: Proceedings of the Second International Symposium on High Performance Computing, ISHPC '99, Springer-Verlag, London, UK, UK, 1999, pp. 121–132.
   URL http://dl.acm.org/citation.cfm?id=646347.690219
9. P. A. Kulkarni, D. B. Whalley, G. S. Tyson, J. W. Davidson, Practical exhaustive optimization phase order exploration and evaluation, ACM Trans. Archit. Code Optim. 6 (1) (2009) 1:1–1:36. doi:10.1145/1509864.1509865.
   URL http://doi.acm.org/10.1145/1509864.1509865
10. P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, D. Jones, Fast searches for effective optimization phase sequences, SIGPLAN Not. 39 (6) (2004) 171–182. doi:10.1145/996893.996863.
    URL http://doi.acm.org/10.1145/996893.996863
11. E. Park, S. Kulkarni, J. Cavazos, An evaluation of different modeling techniques for iterative compilation, in: Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems, CASES '11, ACM, New York, NY, USA, 2011, pp. 65–74. doi:10.1145/2038698.2038711.
    URL http://doi.acm.org/10.1145/2038698.2038711
12. A. Monsifrot, F. Bodin, R. Quiniou, A machine learning approach to automatic production of compiler heuristics, in: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications, AIMSA '02, Springer-Verlag, London, UK, UK, 2002, pp. 41–50.
    URL http://dl.acm.org/citation.cfm?id=646053.677574
13. M. Stephenson, S. Amarasinghe, M. Martin, U.-M. O'Reilly, Meta optimization: improving compiler heuristics with machine learning, SIGPLAN Not. 38 (5) (2003) 77–90. doi:10.1145/780822.781141.
    URL http://doi.acm.org/10.1145/780822.781141
14. M. Tartara, S. Crespi Reghizzi, Continuous learning of compiler heuristics, ACM Trans. Archit. Code Optim. 9 (4) (2013) 46:1–46:25. doi:10.1145/2400682.2400705.
    URL http://doi.acm.org/10.1145/2400682.2400705
15. F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, C. K. I. Williams, Using machine learning to focus iterative optimization, in: Proceedings of the International Symposium on Code Generation and Optimization, CGO '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 295–305. doi:10.1109/CGO.2006.37.
    URL http://dx.doi.org/10.1109/CGO.2006.37

16. R. C. Whaley, A. Petitet, Minimizing development and maintenance costs in supporting persistently optimized BLAS, Software: Practice and Experience 35 (2) (2005) 101–121.
17. Openblas, an optimized blas library (2012).
URL available at http://xianyi.github.com/OpenBLAS/
18. G. Guennebaud, B. Jacob, et al., Eigen v3, http://eigen.tuxfamily.org (2010).
19. Intel, Intel mkl, available at http://software.intel.com/en-us/intel-mkl (2012).
20. J. Bilmes, K. Asanović, C. Chin, J. Demmel, Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology, in: Proceedings of the International Conference on Supercomputing, ACM SIGARC, Vienna, Austria, 1997.
21. K. Goto, R. A. van de Geijn, Anatomy of high-performance matrix multiplication, ACM Trans. Math. Softw. 34 (3) (2008) 12:1–12:25. doi:10.1145/1356052.1356053.
URL http://doi.acm.org/10.1145/1356052.1356053
22. F. G. Van Zee, R. A. Van De Geijn, BLIS: A framework for rapidly instantiating BLAS functionality, ACM Transactions on Mathematical Software 41 (3).
23. T. M. Smith, R. A. van de Geijn, M. Smelyanskiy, J. R. Hammond, F. G. V. Zee, Anatomy of high-performance many-threaded matrix multiplication, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014, 2014, pp. 1049–1059. doi:10.1109/IPDPS.2014.110.
URL http://dx.doi.org/10.1109/IPDPS.2014.110
24. K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, Is search really necessary to generate high-performance blas?, Proceedings of the IEEE 93 (2), special issue on "Program Generation, Optimization, and Adaptation".
25. V. Volkov, J. W. Demmel, Benchmarking gpus to tune dense linear algebra, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC '08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 31:1–31:11.
URL http://dl.acm.org/citation.cfm?id=1413370.1413402
26. R. Nath, S. Tomov, J. Dongarra, An Improved Magma Gemm For Fermi Graphics Processing Units, Int. J. High Perform. Comput. Appl. 24 (4) (2010) 511515. doi:10.1177/1094342010385729.
27. V. I. Kelefouras, A. Kritikakou, C. Goutis, A Matrix–Matrix Multiplication methodology for single/multi-core architectures using SIMD, Supercomputing, Springerdoi:10.1007/s11227-014-1098-9.
28. N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, SIGPLAN Not. 42 (6) (2007) 89–100. doi:10.1145/1273442.1250746.
URL http://doi.acm.org/10.1145/1273442.1250746
29. N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, The gem5 simulator, SIGARCH Comput. Archit. News 39 (2) (2011) 1–7. doi:10.1145/2024716.2024718.
URL http://doi.acm.org/10.1145/2024716.2024718
30. T. Austin, E. Larson, D. Ernst, Simplescalar: An infrastructure for computer system modeling, Computer 35 (2002) 59–67. doi:10.1109/2.982917.
URL http://dl.acm.org/citation.cfm?id=619072.621910
31. S. M. Bhandarkar, H. R. Arabnia, The REFINE Multiprocessor - Theoretical Properties and Algorithms., Parallel Computing 21 (11) (1995) 1783–1805.
32. H. R. Arabnia, J. W. Smith, A reconfigurable interconnection network for imaging operations and its implementation using a multi-stage switching box, 1993, pp. 349–357.
33. M. A. Wani, H. R. Arabnia, Parallel edge-region-based segmentation algorithm targeted at reconfigurable MultiRing network, The Journal of Supercomputing 25 (1) (2003) 43–62.
34. H. R. Arabnia, A parallel algorithm for the arbitrary rotation of digitized images using process-and-data-decomposition approach, Journal of Parallel and Distributed Computing 10 (2) (1990) 188–192.
35. H. R. Arabnia, M. A. Oliver, A transputer network for fast operations on digitised images, Computer graphics forum 8 (1) (1989) 3–11.
36. S. M. Bhandarkar, H. R. Arabnia, The Hough transform on a reconfigurable multi-ring network, Journal of Parallel and Distributed Computing 24 (1) (1995) 107–114.

37. H. R. Arabnia, M. A. Oliver, A transputer network for the arbitrary rotation of digitised images, The Computer Journal 30 (5) (1987) 425–432.
38. H. R. Arabnia, S. M. Bhandarkar, Parallel stereocorrelation on a reconfigurable multi-ring network, The Journal of supercomputing 10 (3) (1996) 243–269.
39. H. R. Arabnia, M. A. Oliver, Arbitrary rotation of raster images with SIMD machine architectures, Computer Graphics Forum 6 (1) (1987) 3–11.
40. S. M. Bhandarkar, H. R. Arabnia, J. W. Smith, A reconfigurable architecture for image processing and computer vision, International journal of pattern recognition and artificial intelligence 9 (02) (1995) 201–229.
41. H. Arabnia, A distributed stereocorrelation algorithm, in: Computer Communications and Networks, 1995. Proceedings., Fourth International Conference on, IEEE, 1995, pp. 479–482.
42. R. C. Whaley, A. Petitet, J. J. Dongarra, Automated empirical optimization of software and the ATLAS project, Parallel Computing 27 (1–2) (2001) 3–35.
43. R. C. Whaley, J. Dongarra, Automatically Tuned Linear Algebra Software, in: Ninth SIAM Conference on Parallel Processing for Scientific Computing, 1999, cD-ROM Proceedings.
44. R. C. Whaley, J. J. Dongarra, Automatically tuned linear algebra software, in: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, Supercomputing '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 1–27.
URL http://dl.acm.org/citation.cfm?id=509058.509096
45. R. C. Whaley, J. Dongarra, Automatically Tuned Linear Algebra Software, Tech. Rep. UT-CS-97-366, University of Tennessee (December 1997).
46. P. Bjørstad, F. Manne, T. Sørevik, M. Vajtersic, Efficient matrix multiplication on simd computers, SIAM J. MATRIX ANAL. APPL 13 (1992) 386–401.
47. R. A. V. D. Geijn, J. Watts, Summa: Scalable universal matrix multiplication algorithm, Tech. rep. (1997).
48. S. Chatterjee, A. R. Lebeck, P. K. Patnala, M. Thottethodi, Recursive array layouts and fast parallel matrix multiplication, in: In Proceedings of Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures, 1999, pp. 222–231.
49. B. Moon, H. V. Jagadish, C. Faloutsos, J. H. Saltz, Analysis of the clustering properties of the hilbert space-filling curve, IEEE Transactions on Knowledge and Data Engineering 13 (2001) 2001.
50. M. Thottethodi, S. Chatterjee, A. R. Lebeck, Tuning strassen's matrix multiplication for memory efficiency, in: In Proceedings of SC98 (CD-ROM, 1998.
51. K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, Is search really necessary to generate high-performance blas?, Proceedings of the IEEE 93 (2).
52. M. Kulkarni, K. Pingali, An experimental study of self-optimizing dense linear algebra software, Proceedings of the IEEE 96 (5) (2008) 832–848.
53. E. Garcia, I. E. Venetis, R. Khan, G. R. Gao, Optimized dense matrix multiplication on a many-core architecture, in: Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II, Euro-Par'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 316–327.
URL http://dl.acm.org/citation.cfm?id=1885276.1885308
54. J. Choi, A new parallel matrix multiplication algorithm on distributed-memory concurrent computers., Concurrency - Practice and Experience 10 (8) (1998) 655–670.
URL http://dblp.uni-trier.de/db/journals/concurrency/concurrency10.html
55. J. Kurzak, W. Alvaro, J. Dongarra, Optimizing matrix multiplication for a short-vector simd architecture - cell processor, Parallel Comput. 35 (3) (2009) 138–150. doi:10.1016/j.parco.2008.12.010.
URL http://dx.doi.org/10.1016/j.parco.2008.12.010
56. F. Desprez, F. Suter, Impact of mixed-parallelism on parallel implementations of the strassen and winograd matrix multiplication algorithms: Research articles, Concurr. Comput. : Pract. Exper. 16 (8) (2004) 771–797. doi:10.1002/cpe.v16:8.
URL http://dx.doi.org/10.1002/cpe.v16:8
57. M. Hattori, N. Ito, W. Chen, K. Wada, Parallel matrix-multiplication algorithm for distributed parallel computers, Syst. Comput. Japan 36 (4) (2005) 48–59. doi:10.1002/scj.v36:4.
URL http://dx.doi.org/10.1002/scj.v36:4

58. S. Hunold, T. Rauber, G. Rünger, Multilevel hierarchical matrix multiplication on clusters, in: Proceedings of the 18th annual international conference on Supercomputing, ICS '04, ACM, New York, NY, USA, 2004, pp. 136–145. doi:10.1145/1006209.1006230. URL http://doi.acm.org/10.1145/1006209.1006230
59. M. Krishnan, J. Nieplocha, Memory efficient parallel matrix multiplication operation for irregular problems, in: Proceedings of the 3rd conference on Computing frontiers, CF '06, ACM, New York, NY, USA, 2006, pp. 229–240. doi:10.1145/1128022.1128054. URL http://doi.acm.org/10.1145/1128022.1128054
60. S. Hunold, T. Rauber, Automatic tuning of pdgemm towards optimal performance, in: Proceedings of the 11th international Euro-Par conference on Parallel Processing, Euro-Par'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 837–846. doi:10.1007/11549468_91.
61. F. Desprez, F. Suter, Impact of Mixed–Parallelism on Parallel Implementations of Strassen and Winograd Matrix Multiplication Algorithms, Rapport de recherche RR-4482, INRIA (2002).
URL http://hal.inria.fr/inria-00072106
62. G. Tsilikas, M. Fleury, Matrix multiplication performance on commodity shared-memory multiprocessors, in: Proceedings of the international conference on Parallel Computing in Electrical Engineering, PARELEC '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 13–18. doi:10.1109/PARELEC.2004.43.
URL http://dx.doi.org/10.1109/PARELEC.2004.43
63. G. Rünger, M. Schwind, Fast recursive matrix multiplication for multi-core architectures, Procedia Computer Science 1 (1) (2010) 67–76, international Conference on Computational Science 2010 (ICCS 2010).
64. M. Krishnan, J. Nieplocha, Srumma: A matrix multiplication algorithm suitable for clusters and scalable shared memory systems, Parallel and Distributed Processing Symposium, International 1 (2004) 70b. doi:http://doi.ieeecomputersociety.org/10.1109/IPDPS.2004.1303000.
65. V. Strassen, Gaussian elimination is not optimal., Numerische Mathematik 14 (3) (1969) 354–356.
66. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. Zhou, Cilk: an efficient multithreaded runtime system, SIGPLAN Not. 30 (8) (1995) 207–216. doi:10.1145/209937.209958.
URL http://doi.acm.org/10.1145/209937.209958
67. P. Michaud, Replacement policies for shared caches on symmetric multicores: a programmer-centric point of view, in: Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, HiPEAC '11, ACM, New York, NY, USA, 2011, pp. 187–196. doi:10.1145/1944862.1944890.
URL http://doi.acm.org/10.1145/1944862.1944890
68. D. S. Nikolopoulos, Code and data transformations for improving shared cache performance on smt processors, in: ISHPC, 2003, pp. 54–69.
69. J. Kurzak, S. Tomov, J. Dongarra, Autotuning gemm kernels for the fermi gpu. 23 (11) (2012) 2045–2057.
URL http://dblp.uni-trier.de/db/journals/tpds/tpds23.html
70. C. Jiang, M. Snir, Automatic tuning matrix multiplication performance on graphics hardware, in: In the proceesings of the 14th International Conference on Parallel Architecture and Compilation Techniques (PACT), 2005, pp. 185–196.
71. J. Li, S. Ranka, S. Sahni, Strassen's matrix multiplication on gpus, in: Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems, ICPADS '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 157–164. doi:10.1109/ICPADS.2011.130.
URL http://dx.doi.org/10.1109/ICPADS.2011.130
72. J. M. Cecilia, J. M. G. 0001, M. Ujaldon, The gpu on the matrix-matrix multiply: Performance study and contributions., in: B. M. Chapman, F. Desprez, G. R. Joubert, A. Lichnewsky, F. J. Peters, T. Priol (Eds.), PARCO, Vol. 19 of Advances in Parallel Computing, IOS Press, 2009, pp. 331–340.
URL http://dblp.uni-trier.de/db/conf/parco/parco2009.html

73. 2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014, IEEE, 2014.
URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6832911
74. T. Athil, R. Christian, Y. B. Reddy, Cuda memory techniques for matrix multiplication on quadro 4000, in: Proceedings of the 2014 11th International Conference on Information Technology: New Generations, ITNG '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 419–425. doi:10.1109/ITNG.2014.24.
URL http://dx.doi.org/10.1109/ITNG.2014.24
75. L. Djinevski, S. Arsenovski, S. Ristov, M. Gusev, Performance drawbacks for matrix multiplication using set associative cache in GPU devices, in: Information & Communication Technology Electronics & Microelectronics (MIPRO), 2013 36th International Convention on, IEEE, 2013, p. 193198.
76. K. Matsumoto, N. Nakasato, T. Sakai, H. Yahagi, S. G. Sedukhin, Multi-level Optimization of Matrix Multiplication for GPU-equipped Systems, Procedia Computer Science 4 (0) (2011) 342351, proceedings of the International Conference on Computational Science, {ICCS} 2011. doi:10.1016/j.procs.2011.04.036.
URL http://www.sciencedirect.com/science/article/pii/S1877050911000949