A Matrix--Matrix Multiplication methodology for single/multi-core architectures using SIMD

KELEFOURAS, Vasileios <http://orcid.org/0000-0001-9591-913X>, KRITIKAKOU, Angeliki and GOUTIS, Costas

Citation:

KELEFOURAS, Vasileios, KRITIKAKOU, Angeliki and GOUTIS, Costas (2014). A Matrix--Matrix Multiplication methodology for single/multi-core architectures using SIMD. The Journal of Supercomputing, 68 (3), 1418-1440. [Article]

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

A Matrix-Matrix Multiplication Methodology for single/multi-core architectures using SIMD

**Vasilios Kelefouras, Angeliki Kritikakou
and Costas Goutis**

Received: date / Accepted: date

Abstract In this paper, a new methodology for speeding up Matrix-Matrix Multiplication using Single Instruction Multiple Data unit, at one and more cores having a shared cache, is presented. This methodology achieves higher execution speed than ATLAS state of the art library (speedup from 1.08 up to 3.5), by decreasing the number of instructions (load/store and arithmetic) and the data cache accesses and misses in the memory hierarchy. This is achieved by fully exploiting the software characteristics (e.g. data reuse) and hardware parameters (e.g. data caches sizes and associativities) as one problem and not separately, giving high quality solutions and a smaller search space.

Keywords Matrix-Matrix Multiplication · data cache · cache associativity · multi-core · SIMD · memory management

1 Introduction

The state of the art (SOA) hand/self-tuning libraries for linear algebra and Fast Fourier Transform (FFT) algorithm, such as ATLAS [54], OpenBLAS [1], GotoBLAS2 [29], Eigen [19], Intel_MKL [24], PHiPAC [4], FFTW [13], and SPIRAL [35], manage to find a near-optimum binary code for a specific application using a large exploration space (many different executables are tested and the fastest is picked). The development of a self-tuning library is a difficult and time-consuming task for two reasons. Firstly, 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. Secondly, the optimum parameters for two slightly different architectures are different. Such a case is Matrix-Matrix

Vasilios Kelefouras
University of Patras
E-mail: kelefouras@ece.upatras.gr

Multiplication (MMM) algorithm, which is a major kernel in linear algebra and also the topic of this paper.

The optimization sub-problems in compilers and MMM are interdependent; this means that by optimizing one sub-problem, another is degraded. These dependencies require that all phases should be optimized together as one problem and not separately. 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) [41] [44] or to apply predictive heuristics [3] [18]. Nowadays compilers and related works, apply either iterative compilation techniques [49] [10] [26] [32], or both iterative compilation and machine learning compilation techniques to restrict the configurations' search space [31] [40] [36] [45] [47] [2]. 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 aforementioned SOA libraries optimize all the above parameters separately by using heuristics and empirical techniques.

The major contributions of this paper are six. Firstly, we introduce a new MMM methodology which is faster than the ATLAS SOA library. Secondly, the optimization is done by fully exploiting the software (s/w) characteristics and the major hardware (h/w) parameters, as one problem and not separately, giving high quality solutions and a smaller search space. Thirdly, the final schedule is found theoretically and not experimentally, according to the input size and to the h/w parameters. Furthermore, this is the first time for this algorithm that the data cache associativity is fully exploited. Also, this is the first time that loop tiling is applied according to the data cache sizes, data cache associativities and the data arrays layouts. At last, the proposed methodology, due to the major contribution of number (ii) above it gives a smaller code size and a smaller compilation time, as it does not test a large number of alternative schedules, as the ATLAS library does.

The proposed methodology is compared with the SOA ATLAS library. It is tested on desktop Personal Computers (PCs) using the Single Instruction Multiple Data (SIMD) unit; the Intel Pentium core 2 duo and i7 processors have been used. Also, the Valgrind [38] tool is used to measure the total number of instructions executed and the number of L1 and L2 data accesses and misses. Although the proposed methodology is written in C language using Streaming SIMD Extension (SSE) intrinsics, it achieves speedup from 1.08 up to 3.5 over ATLAS for one core; if the proposed methodology would be implemented in assembly language a higher speedup would occur (the proposed methodology is at a high level and this is beyond the scope of this paper). The proposed methodology achieves a very large performance gain for small matrices sizes, a large performance gain for medium matrices sizes and a significant gain for large matrices sizes. Although it seems that the MMM optimization problem for small and medium matrices sizes is not important, there are applications that MMM lies inside large loops, and hence high performance for smaller matrices sizes is critical. In this paper, the MMM scheduling for single core CPUs and for multi core CPUs, is explained in detail, making it easy to be implemented by everyone.

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

There are several works optimizing the MMM algorithm for one core [17,16, 29,5,15,8,37,48,56,30]. The aforementioned works use loop-tiling transformation to utilize the data cache hierarchy and the register file. The problem is partitioned into smaller problems (sub-problems) whose smaller matrices (Tiles/sub-matrices) fit in the smaller memories. Although loop-tiling is necessary, tiles are found by taking into account either the cache sizes, or the data arrays layouts; we claim that loop tiling must take into account, the cache sizes, their associativities and the data arrays layouts, together. For example, according to ATLAS, the size of the three tiles (one for each matrix) must be lower or equal to the 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 (the general purpose processors have set-associative caches), they cannot simultaneously fit in data cache duo to the cache modulo effect. In [56], analytical models are presented for estimating the optimum tile size values assuming only fully associative caches, which in practice are very rare. In contrast to the proposed methodology, the above works find the scheduling and the tile sizes 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.

There are several works that optimize the MMM algorithm for many cores [57, 14,9,33,12,21,23,28,22,11,50,42,27] and for GPUs [20,25]. Regarding multicore architectures, the vast majority of the related works, such as SRUMMA [27], deals with the cluster architectures and mainly with how to 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 [27] describes the best parallel algorithm which is suitable for clusters and scalable shared memory systems. About half of the above works, use the Strassen's algorithm [46] 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 [7] or by cblas_sgemm library of ATLAS. Furthermore, [34] and [39] show how shared caches can be utilized.

ATLAS [54], [55], [53], [52], [51], [43], 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 two techniques for supplying different implementations of kernel operations: multiple implementation and code generation. Although ATLAS is one of the SOA library for MMM algorithm, its techniques for supplying different implementations of kernel operations concerning memory management are empirical.

During the installation of ATLAS, 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. Moreover, for ATLAS implementation and tuning, there was access at a wide range of hardware architecture details, such as G4, G5, CoreDuo, and Core2Duo by Apple and UltraSPARC III platform which ATLAS exploited. Also, the proposed methodology lies at a higher level of abstraction than ATLAS because the main features of ATLAS are on the one hand the extremely complex empirical tuning step that is required and on the other hand the large number of compiler options that are used. These two features are beyond the scope of the proposed methodology which is mainly focused on memory utilization. Furthermore, the proposed methodology, ATLAS and all the above SOA libraries use the SIMD unit (the performance is highly increased by executing instructions with 128/256-bit data in each cycle).

3 Proposed Methodology

This paper presents a new methodology for speeding up Matrix-Matrix Multiplication (MMM) using SIMD unit. The proposed methodology achieves high execution speed in single core and multiple core CPUs having a shared cache memory, by fully and simultaneously exploiting the combination of the s/w characteristics (production-consumption, data reuse, MMM parallelism) and h/w parameters, i.e. the number of the cores, the number of data cache memories, the size of each memory, the size and the number of the SIMD registers (XMM/YMM), the associativities of the data caches and the SSE instructions' latencies. For different h/w parameters, different schedules for MMM are emerged.

To utilize the memory hierarchy, the proposed methodology partitions the three matrices into smaller ones (tiles) according to the memory architecture parameters and applies a different schedule for each memory (Subsect. 3.1.1-3.1.3). The number of the levels of tiling depends on the input size and on the data cache sizes and it is found by ineq.1 and ineq.2. Although loop tiling decreases the number of memories accesses, it increases the number of load/store and addressing instructions; however, memory management is MMM performance critical parameter and thus loop tiling is performance efficient.

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

To efficiently use the SSE instructions, ATLAS converts the data layout of B from row-wise to column-wise. The proposed methodology converts the data layout of B from row-wise to tile-wise, i.e. its elements are written into memory just as they are fetched; in this way, the number of data cache misses is decreased. This routine has also been optimized and its execution time is included to the total execution time.

The proposed methodology is presented in Subsect. 3.1 and Subsect. 3.2, when one and all CPU cores are used, respectively.

3.1 Using single CPU core

The proposed methodology partitions the three arrays according to the number of the memories, their sizes and the data cache associativities and it applies a different schedule for each memory.

In the case that there is a two level cache architecture (most cases), the MMM problem is divided into three cases according to the cache sizes and to the input size (the first case holds for small input sizes, the second for medium and the third for large ones):

i. SIMD register file utilization - For small matrices sizes (usually smaller than 90×90 for typical cache sizes), i.e. if all the data of B and the data of 2 rows of A fit in different ways of L1 data cache (ineq. 1), the scheduling given in Subsect. 3.1.1 is used.

$$\frac{L1 \times (assoc - k)}{assoc} \geq M \times P \times element\_size, \tag{1}$$

where $k = \lceil \frac{2 \times M \times element\_size}{L1/assoc} \rceil \leq \frac{assoc}{2}$, $L1$ is the size of the L1 data cache memory in bytes, $element\_size$ is the size of the arrays elements in bytes (e.g. $element\_size = 4$ for floating point numbers) and $assoc$ is the L1 associativity, e.g. for an 8-way L1 data cache, $assoc = 8$. $M \times P$ is the size of the B array in elements. k is an integer and it gives the number of L1 data cache lines with identical L1 addresses used for 2 rows of A; for the reminder of this paper we will more freely say that we use k cache ways for A and $assoc - k$ cache ways for B (in other words A and B are written in separate data cache ways). $assoc \neq 1$ since direct mapped caches do not exist in SIMD architectures ($assoc \geq 8$ in most modern architectures).

Given that each row of A is multiplied by all the columns of B, B is reused N times and thus it has to remain in L1 data cache. To do this, the cache lines of A must be written in L1 without conflict with the B ones. Two rows of A have to fit in L1, the current processed row and the next one for two reasons. Firstly, the next processed row must also be loaded in L1 without conflict with the B ones. Secondly, when the current processed row has been multiplied by all the columns of B, the L1 cache lines containing the previous row of A are replaced by the next processed row ones according to the LRU cache replacement policy, without conflict with the B ones. To fully utilize the L1 data cache and to minimize the number of L1 misses, two rows of A are multiplied by all the columns of B and exact $(assoc - 1)$ cache conflicts occur (if a larger number of conflicts occurs, the data are spilled from L1; if a smaller number of conflicts occurs, only a small part of L1 is utilized). This is achieved by storing the rows of A and the columns of B in consecutive main memory locations and by using $(k \times \frac{L1}{assoc})$ L1 memory size for A and $((assoc - k) \times \frac{L1}{assoc})$ L1 memory size for B (ineq. 1). We can more freely say that this is equivalent to using k cache ways for A and $(assoc - k)$ cache ways for B. 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. It is important to say that if we use $L1 \geq (M \times P + 2 \times M) \times element\_size$ instead of ineq. 1, the number of L1 misses will be much larger because A and B cache lines would conflict with each other. To our knowledge, this is the first time for MMM algorithm that the cache associativity is utilized.

The C array is stored into main memory infrequently (usually 1 cache line is written to memory when 1 entire row of A has been multiplied by 4 columns of B) and thus the number of conflicts due to C can be neglected (victim cache if exists, eliminates the misses of C).

ii. L1 data cache and SIMD register file utilization - For medium matrices sizes where ineq. 1 does not hold, another schedule is used. If all the data of A and a Tile1 of B fit in separate ways of L2 cache (ineq. 2), the scheduling given in Subsect. 3.1.2 is used.

$$\frac{L2 \times (assoc - 1)}{assoc} \geq N \times M \times element\_size \qquad (2)$$

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). $N \times M$ is the size of array A in elements.

The size of A is much larger than the size of a Tile1 of B and thus $((assoc - 1) \times \frac{L2}{assoc})$ and $(\frac{L2}{assoc})$ L2 size is needed for A and B-C arrays, respectively. In most architectures the size of one Tile1 of B (suppose $Tile1$) is smaller than one L2 way since i) the associativity of L2 cache is always larger than 8, i.e. $assoc \geq 8$, ii) $L2 \geq 8 \times L1$ and iii) $L1 \succ Tile1$.

iii. L2, L1 and SIMD register file utilization - For large matrices sizes (usually larger than 900 × 900 for typical cache sizes), where all the data of A and a Tile1 of B, do not fit in L2 cache, i.e. ineq. 2 does not hold, the scheduling given in Subsect. 3.1.3 is used.

For arrays sizes that are close to two of the above cases (i, ii and iii), both solutions are tested and the fastest is picked.

Most of the current general purpose processors contain separate L1 data and instruction caches and thus we can assume that shared/unified caches, contain only data. This is because the MMM code size is small and it fits in L1 instruction cache here.

The schedules which utilize the number of XMM/YMM registers, L1 and L2 sizes, corresponding to the three cases above, are given in the Subsect. 3.1.1, Subsect. 3.1.2 and Subsect. 3.1.3, respectively. In Subsect. 3.1.4, we give the schedule where B data layout is transformed from row-wise to tile-wise.

3.1.1 SIMD register file utilization

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. 3).

$$R = m + 1 + 1 \tag{3}$$

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

For small matrices sizes (ineq. 1 holds), each row of A is multiplied by all columns of B, optimizing the L1 data cache size and associativity (Fig. 1 where $p1 = P$).

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. 2). 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. 1, Fig. 2). 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. 2); the intermediate results in XMM2:XMM7 registers, are always produced and consumed. When the 1st row of A has been fully 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;

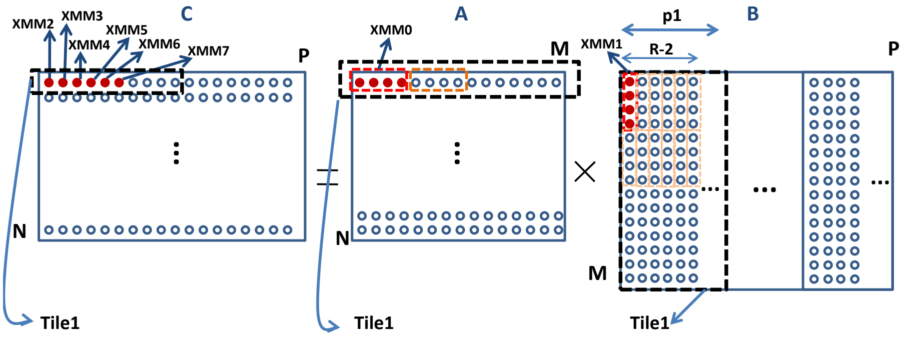
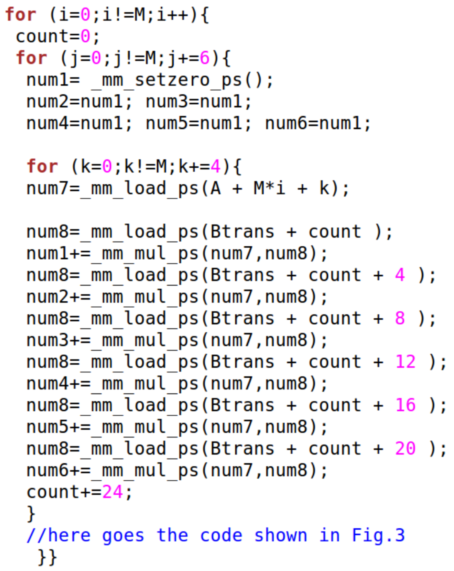


Fig. 1 The proposed methodology for one core when tiling for L1 data cache is used



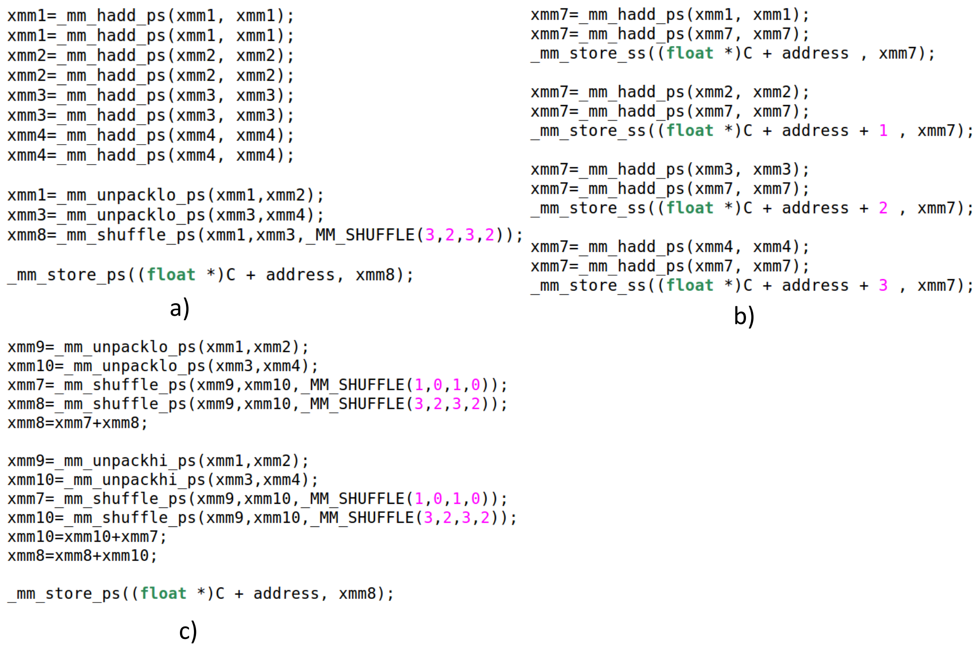
```
for (i=0;i!=M;i++){
 count=0;
 for (j=0;j!=M;j+=6){
  num1= _mm_setzero_ps();
  num2=num1; num3=num1;
  num4=num1; num5=num1; num6=num1;

  for (k=0;k!=M;k+=4){
  num7=_mm_load_ps(A + M*i + k);

  num8=_mm_load_ps(Btrans + count );
  num1+=_mm_mul_ps(num7,num8);
  num8=_mm_load_ps(Btrans + count + 4 );
  num2+=_mm_mul_ps(num7,num8);
  num8=_mm_load_ps(Btrans + count + 8 );
  num3+=_mm_mul_ps(num7,num8);
  num8=_mm_load_ps(Btrans + count + 12 );
  num4+=_mm_mul_ps(num7,num8);
  num8=_mm_load_ps(Btrans + count + 16 );
  num5+=_mm_mul_ps(num7,num8);
  num8=_mm_load_ps(Btrans + count + 20 );
  num6+=_mm_mul_ps(num7,num8);
  count+=24;
  }
  //here goes the code shown in Fig.3
   }}
```

Fig. 2 MMM C code using SSE intrinsics for the 3.1.1 case

three of them are shown in Fig. 3, 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. 3-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. 3-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. 3-c), unpacks the 32-bit values of the four registers and packs them



```
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. 3 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.

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.

For small matrices sizes (ineq.1 holds), the above schedule, i.e. the schedule with the optimum production-consumption of C (each row of A is multiplied by several columns of B), is the optimum here. is the optimum. We found this schedule theoretically and not experimentally, by exploiting the s/w characteristics and the h/w parameters. This is because each register of C contains more than one C values which have to be added, unpacked and stored into memory; thus, when the production-consumption is maximized, the number of SSE instructions is minimized. Even if the arrays sizes are very large and k iterator in Fig. 2 or equivalent M dimension in Fig. 1, has to be tiled, the tile with the larger tile size possible in M dimension is selected, to decrease the number of SSE instructions (more details in Subsect. 3.1.3). Furthermore, the data reuse of A is the largest possible according to the number of XMM/YMM registers; each intermediate result of C is produced-consumed M times and each element of A is reused $R-2$ times.

3.1.2 L1 data cache utilization

For medium matrices sizes (ineq. 2 holds), another schedule is used. The B array is partitioned into Tile1 tiles of size $M \times p1$ (Fig. 1). The L1 and L2 cache sizes and their associativities are fully exploited.

In general, if the arrays data do not fit in L1 data cache, tiling is applied to decrease the number of L1 misses. Although we could, we do not select the schedule achieving the minimum number of L1 data cache misses here, but we select the schedule achieving the lower number of L1 misses satisfying that the minimum number of L2 misses is achieved, i.e each array is loaded/stored from/to main memory just once and the minimum number of SSE and addressing instructions is achieved. The sub-problems of finding the minimum number of L1 misses, L2 misses and number of instructions, are interdependent; no feasible schedule simultaneously optimizes these 3 subproblems. This is why the proposed methodology optimizes all these sub-problems as one problem and not separately. As we have experimentally been observed (experimental results section - Table 1), for medium input sizes, ATLAS achieves a lower number of L1 misses than the proposed methodology; however, the proposed methodology achieves higher performance for the reasons explained above.

To decrease the number of L1 data cache misses, the largest $p1$ size is selected so that 2 rows of A and one Tile1 of B ($M \times p1$) fit in separate ways of L1 data cache (ineq. 4 and Fig. 1). Since ineq. 2 holds, there is a $p1$ value where $p1 \geq (R-2)$ and 2 rows of A and $p1$ columns of B fit in L1 data cache; this is because the L2 size is restricted by the L1 size. We select the maximum $p1$ size giving ineq. 4; $p1$ is a multiple of $(R-2)$.

$$\frac{L1 \times (assoc - k)}{assoc} \geq p1 \times M \times element\_size \tag{4}$$

where $k = \lceil \frac{2 \times M \times element\_size}{L1/assoc} \rceil \leq \frac{assoc}{2}$, $L1$ is the size of the L1 data cache memory, M is the size of each row of A in elements, $element\_size$ is the size of the arrays elements in bytes and $assoc$ is the associativity, e.g. for an 8-way data cache, $assoc = 8$. k value gives the number of L1 cache lines with identical L1 address used by array A. If $p1$ does not equally divide M, padding instructions are needed and performance may be degraded; $p1$ sizes that equally divide M are preferable.

Each Tile1 of B ($M \times p1$) is reused N times (it is multiplied by N rows of A) and thus it has to remain in L1 data cache (all the elements of Tile1 of B are written in consecutive memory locations). To do this, two rows of A (the current processed row and the next one) must be written in L1 data cache without conflict with the Tile1 of B. Furthermore, the two rows of A and the Tile1 of B must be written in cache having k and $(assoc - k)$ identical L1 addresses, respectively for minimizing the number of the L1 misses. In most cases, $assoc = 8$ and two rows of A occupy size that equals to 1 or 2 L1 ways while the Tile1 of B occupies size that equals to 6 or 7 L1 ways, respectively.

Regarding C array, the C cache lines are fetched infrequently (one cache line is written to memory when 1 entire row of A is multiplied by several columns of B) and thus the number of conflicts due to C can be neglected (victim cache eliminates the misses of C). This is the first time for MMM that the data cache associativity is fully utilized.

The schedule is shown in Fig. 1. Each row of A is multiplied by the $(R-2)$ columns of the first Tile1 of B, exactly as in Subsect. 3.1.1, utilizing the number of XMM/YMM registers. Then, the same row of A as above, is multiplied by the next $(R-2)$ columns of the first Tile1 of B etc (Fig. 1). This is repeated until the first row of A has been multiplied by all the columns of the first Tile1 of B. Then, the remaining $N-1$ rows of A are multiplied one after another by the same Tile1 of B as above. After the first Tile1 of B has been multiplied by all rows of A the next one is fetched and it is again multiplied by all the rows of A, etc (A is loaded from L2 data cache not from main memory). The optimum data reuse of B and the sub-optimum data reuse of A are selected according to the L1 data cache size and associativity. Regarding L1 data cache accesses, the C, A and B arrays are accessed 1, $P/p1$ and 1, respectively. Regarding main memory data accesses, the C, A and B arrays are loaded/stored just once.

As it has already been mentioned, we do not select the schedule achieving the minimum number of L1 data cache misses here, but the schedule achieving the lower number of L1 misses satisfying that each array is loaded/stored from/to main memory just once and the minimum number of SSE and addressing instructions is achieved. All arrays are fetched just once from main memory since the whole A is chosen to fit in L2 data cache (ineq. 2) and the Tile1 of B is chosen to fit in L1 data cache(ineq. 4). Moreover, the minimum number of SSE instructions (unpack, shuffle, add and load/store instructions) is achieved because the M dimension has not been tiled and the minimum number of addressing instructions is achieved because only one of the three loops is tiled. Both are shown in experimental results section (Table 1).

3.1.3 L2 data cache utilization

For large arrays sizes, ineq.2 cannot give a $p1$ value where $p1 \geq (R-2)$ and thus to decrease the number of main memory accesses, the arrays are further partitioned into Tile2 tiles (Fig. 4). The optimum data reuse of A and the sub-optimum data reuse of B is achieved according to the L2 data cache size and associativity. The number of SIMD instructions is the minimum since the tile sizes are as large as possible in the M dimension and the number of the addressing instructions is the minimum because only one of the three loops is tiled here.

Now, instead of multiplying rows of A by rows of B, sub-rows of size $m2$ are multiplied. The matrix A is partitioned into Tile2 tiles of size $n2 \times m2$ (Fig. 4). Also, the Tile1 of B changes its size to $m2 \times p1$. If ineq. 4 cannot give a $p1$ value where $p1 \geq (R-2)$ and $p1$ is a multiple of $R-2$, the largest $p1$ value for $m2 = M/2$ is selected that $p1 \geq (R-2)$, ineq. 5. If ineq. 5, still cannot give a $p1$ value that $p1 \geq (R-2)$, $m2 = M/3$ is selected and etc.

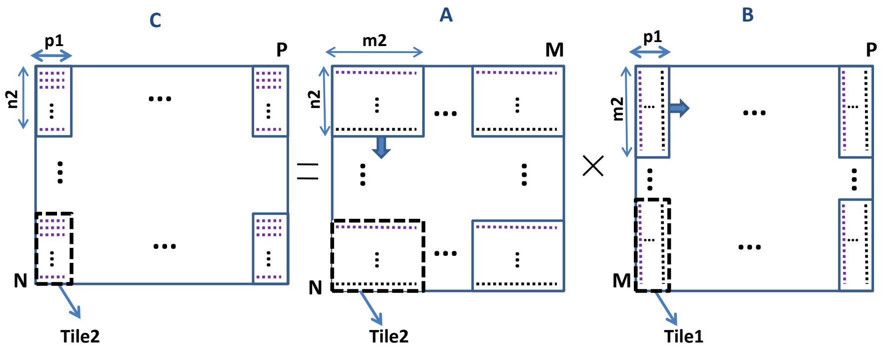


Fig. 4 The proposed methodology for one core when tiling for L2 and L1 data cache is used

$$\frac{L1 \times (assoc - k)}{assoc} \geq p1 \times m2 \times element\_size \tag{5}$$

where $k = \lceil \frac{2 \times m2 \times element\_size}{L1/assoc} \rceil \leq \frac{assoc}{2}$, $m2 = \frac{M}{1}, \frac{M}{2}, ..., \frac{M}{n}$ and n is positive integer ($n \geq 1$).

Two sub-rows of A and $p1$ sub-columns of B of size $m2$, have to fit in separate ways of L1 data cache. It is important to say that ineq. 5 holds only if the tiles elements of A and B are written in consecutive main memory locations (explain further below); otherwise, the tiles sub-rows/sub-columns will conflict with each other due to the cache modulo effect.

The smallest n value is selected since the larger the size of $m2$, the lower the number of the SIMD arithmetic and load/store instructions (the registers containing the C values, are unpacked, added and loaded/written from/to memory less times). However, there are cases that the second smaller n value is selected since it exploits better the L1 data cache size. For example, suppose that there is an L1 data cache of size 32 kbyte, 8-way set associative (each way is 4096 bytes). If $m2 = 600$, $k = \lceil 1.17 \rceil = 2$, two and six L1 ways are used for A and B with exploitation ratios of $\frac{4800}{8192}$ and $\frac{19200}{24576}$, respectively (the larger the exploitation ratio the larger the cache utilization). If $m2 = 400$, $k = \lceil 0.78 \rceil = 1$, the corresponding exploitation ratios are of $\frac{3200}{4096}$ and $\frac{25600}{28672}$, respectively. If $m2 = 400$, a larger number of addressing and load/store instructions (relative to $m2 = 600$) exist, thereby enabling the achievement of larger L1 exploitation rations; however, the actual performance depends on the hit latency memory values and the SSE instructions' latencies.

To efficiently use the L2 cache, the array A is further partitioned into $Tile2$ tiles. A $Tile2$ tile of A (size of $n2 \times m2$), a tile1 tile of B (size of $m2 \times p1$) and a Tile2 of C (size of $n2 \times p1$), have to fit in L2 cache (ineq. 6). Array A uses $assoc - 1$ L2 ways while B-C arrays use only one L2 way. This is because the sum of the sizes of a Tile2 of C and a Tile1 of B, is smaller than one L2 way and their elements are not reused and thus there is no problem if their cache lines are replaced (Tile1 of B is reused in L1 data cache not in L2).

$$\frac{L2 \times (assoc - 1)}{assoc} \geq n2 \times m2 \tag{6}$$

Concerning the data layout of A, when M dimension is tiled, i.e. $m2 \prec M$, 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. 5 and ineq. 6 cannot give a minimum number of cache conflicts since the sub-rows of A will conflict with each other.

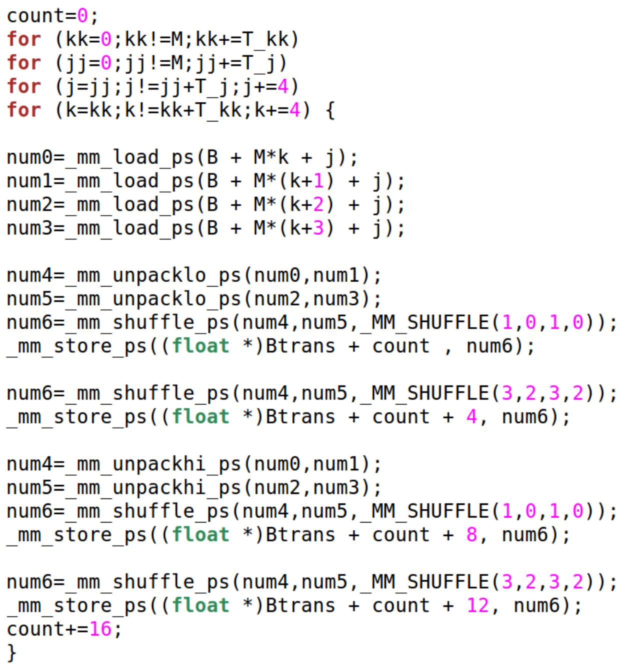
The scheduling is shown in Fig. 4. The first Tile2 of the first Tile2 block column of A is multiplied by all the Tile1 of the first Tile1 block row of B, exactly as in Subsect. 3.1.2. Then, the second Tile2 of the first Tile2 block column of A is multiplied by the same Tile1 tiles as above etc. The procedure ends when all Tile2 block columns of A have been multiplied by all Tile1 block rows of B.

For large matrices sizes, the above schedule is the optimum. This schedule has been found theoretically and not experimentally, by exploiting the s/w characteristics and the h/w parameters. We select the schedule with the optimum data reuse of A, since having Tile1 tiles of B in L1 data cache, they need to be multiplied by as many rows of A as possible before they are spilled in upper level memories.

It is important to say, that apart from the MMM process, there are other processes executed by the cores too. They are Operating System (OS) kernel processes or even user applications processes, such as web and mail browsers or document editors. All these processes including MMM, use the same h/w resources; hence during the running time of MMM, they share the cache, changing its data. Furthermore, for multi-core architectures there are complex contention aware OS schedulers that manage the h/w resources [58] among the cores which effect data motion in shared cache. The point is that we do not know the available shared cache size and the number of its available ways for exploitation, for achieving high performance. Therefore, the L2 cache size used by ineq. 6, is found experimentally.

3.1.4 Scheduling for changing the data array layout

When high performance for MMM is needed, apart from the MMM routine, we must also speed up the routine that changes the data arrays layouts. The B is not written in main memory column wise as ATLAS does, but tile-wise, i.e. its elements are written into memory just as they are fetched (Fig. 5). Firstly, the top four elements of the first, second, third, fourth, fifth and sixth column of B are written to memory, i.e. $(B(0:3,0))$, $(B(0:3,1))$, ..., $(B(0:3,5))$ respectively; then the next top four elements of the first six columns of B are written to memory, i.e. $(B(4:7,0))$, $(B(4:7,1))$, ..., $(B(4:7,5))$ respectively, until $m2$ elements of B have been processed. Afterwards, the procedure continues with the next $R-2$ columns of B etc. In this way, the number of cache misses is decreased as all the elements are accessed in-order.



```
count=0;
for (kk=0;kk!=M;kk+=T_kk)
for (jj=0;jj!=M;jj+=T_j)
for (j=jj;j!=jj+T_j;j+=4)
for (k=kk;k!=kk+T_kk;k+=4) {

num0=_mm_load_ps(B + M*k + j);
num1=_mm_load_ps(B + M*(k+1) + j);
num2=_mm_load_ps(B + M*(k+2) + j);
num3=_mm_load_ps(B + M*(k+3) + j);

num4=_mm_unpacklo_ps(num0,num1);
num5=_mm_unpacklo_ps(num2,num3);
num6=_mm_shuffle_ps(num4,num5,_MM_SHUFFLE(1,0,1,0));
_mm_store_ps((float *)Btrans + count , num6);

num6=_mm_shuffle_ps(num4,num5,_MM_SHUFFLE(3,2,3,2));
_mm_store_ps((float *)Btrans + count + 4, num6);

num4=_mm_unpackhi_ps(num0,num1);
num5=_mm_unpackhi_ps(num2,num3);
num6=_mm_shuffle_ps(num4,num5,_MM_SHUFFLE(1,0,1,0));
_mm_store_ps((float *)Btrans + count + 8, num6);

num6=_mm_shuffle_ps(num4,num5,_MM_SHUFFLE(3,2,3,2));
_mm_store_ps((float *)Btrans + count + 12, num6);
count+=16;
}
```

Fig. 5 C code using SSE intrinsics that changes the data layout of B according to Fig. 4 (rectangles of size 4 × 4 are used)

To change the data layout of B we use the SSE instructions. The array is partitioned into rectangles of size *length* × $(R - 2)$, where *length* is the number of the elements the XMM/YMM registers contain. All these rectangles are fetched one by one, column wise, they are inversed and all the inversed rectangles are written to memory in order creating a new array with tile-wise data layout. The most efficient way (for most architectures) to transpose B when rectangles of size 4 × 4 are used, is that shown in Fig. 5.

3.2 Using multiple CPU cores

When using multiple CPU cores, the MMM problem is partitioned into smaller MMM sub-problems. Each sub-problem corresponds to a thread and each thread is executed in one core only. Each thread must contain more than a specific number of instructions for the cores to be idle as less as possible, since partitioning the MMM into small sub-problems the threads initialization and synchronization time is made comparable to the threads execution time, leading to low performance. Thus, an additional constraint is introduced here.

Most multi core processors, typically contain 2 or 3 levels of cache, having either separate L1 data and instruction caches and a shared L2 cache or separate L1 data and instruction caches, separate unified L2 caches and a shared L3 cache, respectively. All the cache memories have LRU replacement

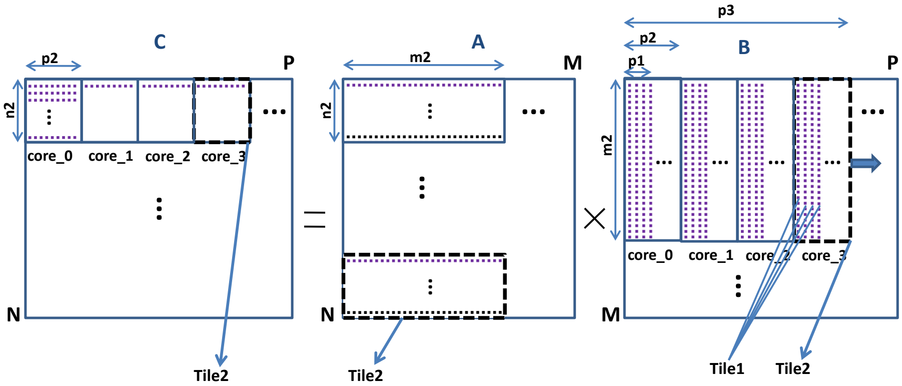


Fig. 6 The proposed methodology for 4 cores having a shared L2 cache. Only the first Tile2 tiles are shown here.

policy. 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 Scheduling when L2 shared cache exists

To utilize L2 shared cache, we partition the three arrays into Tile2 tiles (Fig. 6). Arrays A, B and C Tile2 tiles are of size $n2 \times m2$, $m2 \times p2$ and $n2 \times p2$, respectively (Fig. 6). Each multiplication between two Tile2 tiles creates a different thread. Each multiplication between two Tile2 tiles is made as in Subsect.3.1.1. 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. 6). Thus, $p3$ ($p3 = q \times p2$) 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. 7). Tile2 of A is reused $P/p1$ times and thus it remains in L2 cache. On the other hand, the B-C cache lines used for the multiplication of a Tile2 of A, are of size much smaller than an L2 way and also they are not reused in L2 cache; thus they do not need more cache space than q ways in order not to conflict with the A ones (B cache lines are reused in L1 data cache and thus there is no problem when these cache lines of C conflict with the B ones in L2).

Regarding L2 data cache, the largest $n2$ value which satisfy ineq. 7 is selected (Fig. 6). 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.

$$\frac{L2 \times (assoc - q)}{assoc} \geq n2 \times m2 \tag{7}$$

where the $m2$ value is determined according to L1 data cache size (ineq. 5). $p2$ value depends on the number of instructions each thread must contain and it is found experimentally; if $p2$ 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 8 in most architectures. As explained in the previous subsection, we do not know the exact cache size and the exact number of cache ways to exploit. Therefore, the $L2$ and $assoc$ values in ineq. 7 are found experimentally.

The illustration example consists of a scenario that there are four cores (Fig. 6). Each Tile2 of A is multiplied by a Tile2 of B exactly as in Subsect. 3.1.3. Each multiplication between two Tile2 tiles makes a different thread and each thread is fully executed at only one core. Firstly, 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 ($M/p2$ different threads); all $M/p2$ 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, $N/n2$ and $M/m2$ times, respectively. Concerning L2 shared cache, we select the optimum data reuse of A and the sub-optimum data reuse of B.

In the case that the arrays sizes are small and the Tile2 tiles are larger than the matrices, dividing the MMM problem into threads may decrease performance, since the threads do not achieve the minimum number of instructions needed. Also, in the case that the arrays sizes are small and $p3 = M$, the number of threads (q) executed in parallel is small. In this case, it may be preferable to decrease the data reuse of Tile2 of A in L2 and increase the number of the threads run in parallel, i.e. all Tile2 tiles of each block column of A are multiplied by all Tile2 tiles of the corresponding Tile2 block row of B, in parallel.

3.2.2 Scheduling when L3 shared cache exists

To utilize L3 cache, A and B arrays are further partitioned into Tile3 tiles, of size $((q \times n2) \times m2)$ and $(m2 \times p3)$, respectively (Fig. 7). 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.3.1.1. We determine the Tile1, Tile2 and Tile3 parameters by the data cache sizes and associativities.

Regarding L1 data cache, we compute the values of $m2$ and $p1$ according to the L1 parameters defined by ineq. 5.

Regarding L2 cache, we compute the $n2$ value according to the L2 parameters defined by ineq. 8.

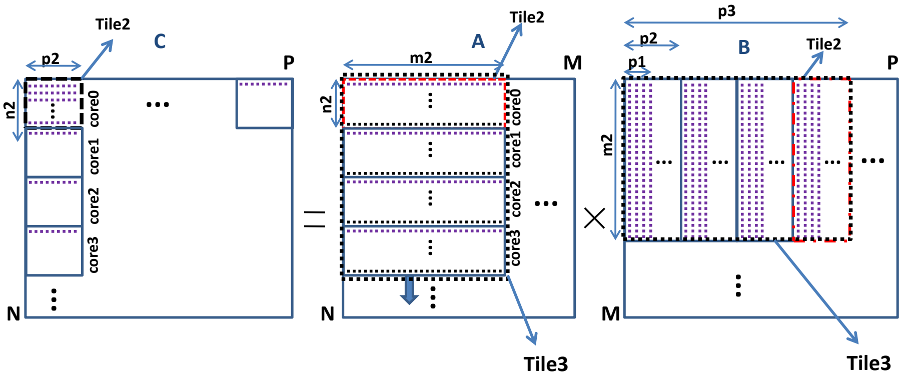


Fig. 7 The proposed methodology for 4 cores having a shared L3 cache. Only the first Tile3 tiles are shown here.

$$\frac{L2 \times (assoc - 1)}{assoc} \geq n2 \times m2 \tag{8}$$

The largest $n2$ value is selected satisfying ineq. 8. $p2$ is found experimentally since each thread has to contain a minimum number of instructions

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). The size of one Tile1 of B (suppose $Tile1$) is always smaller than one L2 way since i) the associativity of L2 cache is always larger than 8, i.e. $assoc \geq 8$, ii) $L2 \geq 8 \times L1$ and iii) $L1 \succ Tile1$.

Regarding L3 cache, we compute $p3$ 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. 9).

$$((assoc - k - 1) \times \frac{L3}{assoc}) \geq m2 \times p3 \tag{9}$$

where $k = \lceil \frac{q \times n2 \times m2}{L3/assoc} \rceil$ and L3 is the size of the L3 cache size exploited. The larger $p3$ value is selected satisfying ineq. 9. Also, $p3 = l \times p2$ where l is an integer. $p2$ value depends on the number of instructions, each thread must contain and it is found experimentally. $m2$ value is found according to L1 data cache size and it is given by ineq. 5.

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 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 illustration example consists of the scenario that there are 4 cores (Fig. 7). Each Tile2 of A is multiplied by a Tile2 of B exactly as in Subsect. 3.1.3. Each multiplication between two Tile2 tiles makes a different thread and each thread is executed in one core only. Firstly, all Tile2 tiles of the first Tile3 of A are multiplied by all Tile2 tiles of the first Tile3 of B ($(p3/p2) \times q$ 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 $N/n2$ times (this is why the Tile3 of B has to fit in L3 shared cache) and each Tile2 of A is reused $(p3/p1)$ 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.

In contrast to the scheduling described in Subsect. 3.2.1 (in which a shared L2 cache exists), we do not multiply each Tile2 of A by the q Tile2 of B in parallel. Instead, we multiply the q Tile2 tiles from A by the several Tile2 tiles from B. This is because each processor has its private L2 cache here, which is used to store a Tile2 of A. Thus, it is memory efficient to reuse the data of this tile as much as possible; each Tile2 of A is loaded into its separate L2 and is reused $(p3/p1)$ times. Also, the largest $p3$ value is selected according to the L3 cache size since the number of main memory accesses depends on this value. A is loaded $N/p3$ times from main memory while B is loaded just once.

However, this schedule may not be performance efficient when $p3 = p2 = P$, e.g. for small matrices sizes, since the memory utilization is lost by the small number of threads executed in parallel. There are only q threads executed in parallel and hence performance may be degraded (synchronization time is large). In this case, it may be preferable to decrease the data reuse of Tile3 of A in L3 and increase the number of threads executed in parallel, i.e. all Tile2 tiles of each block column of A are multiplied by all Tile2 tiles of the corresponding Tile2 block row of B, in parallel.

4 Experimental Results

The experimental results for the proposed methodology, presented in this section, were carried out with a Pentium Intel core 2 duo E6550 and with a Pentium Intel i7-2600K at 3.4Ghz both using SSE instructions and ATLAS 3.8.3 and 3.8.4 respectively. Also Valgrind tool is used to measure the total number of instructions executed and the number of L1 and L2 data accesses and misses [38]. The first processor contains 8 128-bit XMM registers, L1 data and instruction caches of size 32 kbytes and shared L2 cache of size 4 Mbytes. The second processor contains 16 256-bit YMM registers, L1 data and instruction caches of size 32 kbytes, L2 unified cache of size 256 kbytes and shared L3 cache of size 8 Mbytes. At both processors, the Operating system Ubuntu and the gcc-4.4.3 compiler are used. In the experimental procedure, floating point numbers (4 bytes) as elements, were used. 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 layout is always included to the execution times.

Firstly, a performance comparison is made using Pentium Intel core 2 duo E6550 using one of the two cores. It is important to say that using only one core, the thread has to be manually assigned to the 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_sgemm library (Fig. 8). In Fig. 8, the average execution time among 10 executions is shown; there is a significant deviation at ATLAS execution time (up to 15%); this is because different schedules take place for a certain array size.

By using the scheduling provided in Subsect. 3.1.1 on input sizes from $N = 48$ to $N = 72$ (Fig. 8), the proposed methodology has a very large performance gain over ATLAS, i.e. 3.0 to 3.5 times faster. This is because the proposed methodology achieves about 42% less load/store and 58% less arithmetic instructions (Table 1). The number of instructions is less because the number of XMM registers has been fully exploited; 6, 1 and 1 XMM registers used for C, A and B arrays, respectively.

By using the scheduling provided in Subsect. 3.1.2 on input sizes from $N = 120$ to $N = 912$ (Fig. 8), the proposed methodology has a large performance gain, i.e. 1.21 to 3.03 times faster. This is because the proposed methodology executes a smaller number of instructions and it also achieves a smaller number of L1 and main memory data accesses over ATLAS (Table 1). However, ATLAS achieves a smaller number of L1 misses; this is because the proposed methodology achieves the minimum number of L1 data cache misses possible, provided that the minimum number of L2 misses is achieved. In contrast with ATLAS, the proposed methodology optimizes these two sub-problems together as one problem and not separately. Furthermore, the number of SIMD instructions (arithmetic and load/store) is smaller than the ATLAS ones because the arrays are not tiled along to the M dimension (Fig. 4).

By using the scheduling provided in Subsect. 3.1.3 on input sizes from $N = 1200$ to $N = 4800$ (Fig. 8), the proposed methodology achieves a significant but not large performance gain, i.e. 1.08 to 1.14 times faster. In this case, very large arrays exist and one of the performance critical parameters is the number of main memory accesses. The proposed methodology achieves about 3 times less main memory accesses but about 3 times more L2 data cache accesses; main memory is many times slower than L2 and this is why performance gain over ATLAS is achieved. Also, arrays are tiled along to the M dimension too here and thus the number of arithmetical and load/store instructions is almost equal with the ATLAS one (Table 1). By using the scheduling provided in

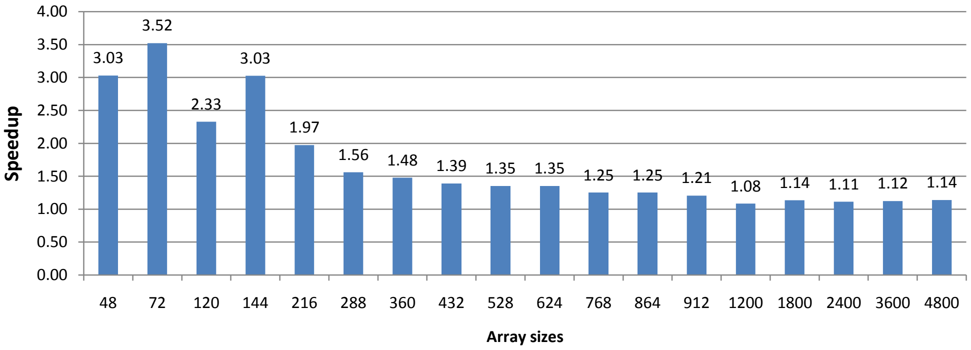


Fig. 8 Speedup of the proposed methodology over cblas_sgemm routine of ATLAS at one core. Square matrices of size $N \times N$ are used here.

Subsect. 3.1.3, the data of C are loaded/written from/to memory $M/m2$ times; the data are fetched from memory, added with the new C values and then they are written back to memory. Each time a C load instruction occurs, its data are fetched from main memory having a large latency penalty. Thus, s/w prefetch instructions can be used to hide the main memory latency. The programmer has to write these instructions before the unpack and shuffle instructions, i.e. at the beginning of the code of Fig. 3.

A performance comparison is also made by using more than one cores. ATLAS optimized library is not supported for many CPU cores, but for one (there is ScaLAPACK [6] which runs at many CPUs - distributed memory). Thus, a performance comparison is done over cblas_sgemm. Given that cblas_sgemm gives the best execution time for one core (suppose ex_time), the best execution time for q cores is always larger than (ex_time/q) as the MMM threads must be synchronized and initialized. Furthermore, the programmer has to set the CPU thread affinity flag for each thread. Otherwise, the OS will make the core assignment, and it will toggle the threads among the cores degrading performance because of the pure data locality.

Firstly, the 2 cores of the core 2 duo are used; a large core utilization factor is achieved, according to the arrays sizes (Table. 2). For $N = 528$, its value is 1.73, while for larger sizes it is near optimum. The core utilization factor is smaller for sizes $N \prec 528$, as the thread initialization time is comparable with its execution time. The utilization factor values are increased for larger matrices sizes for the same reason. The speedup values over ATLAS are shown in Table. 2 (1.99 to 2.11). As it is explained in the next paragraph, if we run ATLAS routine in the two cores, the utilization factor values are much smaller than the proposed methodology ones and the speedup is about 1.2 (last two columns in Table 2).

The proposed methodology is also compared with SRUMMA [27]; SRUMMA achieves the highest execution speed for a large number of multi core CPUs, by well reducing the communication contention among the CPUs. Although

Table 1 Total number of instructions (arithmetic and load/store) and L1, L2 accesses-misses values of the proposed methodology and cblas_sgemm routine of Atlas using one core (Valgrind tool is used)

Size	Instructions (total)		L1 accesses - load/stores		L1 misses / L2 accesses		L2 misses / MM accesses	
	Proposed	Atlas	Proposed	Atlas	Proposed	Atlas	Proposed	Atlas
72	1.64×10^6	2.46×10^6	8.43×10^5	1.2×10^6	4.2×10^3	3.9×10^3	2.5×10^3	2.1×10^3
120	4.5×10^6	5.7×10^6	2.3×10^6	3.0×10^6	1.21×10^4	1.48×10^5	4.89×10^3	5.83×10^3
288	3.45×10^7	4.22×10^7	1.63×10^7	1.95×10^7	1.36×10^5	1.13×10^5	2.2×10^4	2.3×10^4
360	6.09×10^7	7.14×10^7	2.7×10^7	3.16×10^7	3.0×10^5	2.22×10^5	3.36×10^4	3.59×10^4
528	1.6×10^8	1.94×10^8	7.1×10^7	8.07×10^7	1.0×10^6	6.34×10^5	8.8×10^4	1.07×10^5
1200	1.88×10^9	1.80×10^9	6.86×10^8	6.63×10^8	1.9×10^7	6.1×10^6	8.13×10^5	1.9×10^6
1800	6.0×10^9	5.7×10^9	2.1×10^9	2.0×10^9	6.39×10^7	1.9×10^7	2.0×10^6	6.0×10^6
2400	1.39×10^{10}	1.31×10^{10}	4.7×10^9	4.5×10^9	1.5×10^8	4.5×10^7	4.3×10^6	1.3×10^7

Table 2 Core utilization factor and speedup values over cblas_sgemm and SRUMMA, using 2 and 4 cores

	4 cores		2 cores			
array size	core util. factor	speedup over Atlas (one core)	core util. factor	speedup over Atlas (one core)	SRUMMA core util. factor	speedup over SRUMMA / ATLAS on two cores
528	3.56	4.10	1.73	1.99	1.66	1.20
900	3.71	4.07	1.85	2.05	1.67	1.23
1200	3.76	4.15	1.82	2.02	1.67	1.21
1800	3.79	4.11	1.88	2.01	1.72	1.17
2400	3.82	4.12	1.84	2.08	1.74	1.21
3600	3.81	4.21	1.90	2.10	1.91	1.12
4800	3.82	4.24	1.89	2.12	1.76	1.21

SRUMMA and other related work such as [23], minimize the communication contention, they do not optimize the MMM problem for one CPU. SRUMMA partitions the MMM problem into smaller sub-problems, according to the number of the cores (memories sizes are not taken into account) and each core runs the cblas_sgemm ATLAS optimized library. However, to achieve optimum performance, the memories sizes and the data reuse have to be taken into account. This is why SRUMMA core utilization factors are small here (Table. 2). SRUMMA scheduling details are not given in [27] and thus to implement SRUMMA for 1 dual core CPU, we partitioned each one of the three matrices into two and four parts; also, the tiles are multiplied by using both a block row-wise schedule and a block column-wise schedule, and the best core utilization factor value is picked (cblas_sgemm routine was used for each thread).

Large utilization factor values are also achieved using the 4 cores of Pentium Intel i7-2600K at 3.4Ghz (Table. 2). For small arrays sizes, small core utilization factors values are achieved, while for larger ones, the values are near optimum. It is important to say that in both CPUs, performance is highly affected by XMM/YMM and L1 tile selection; smaller or larger tiles sizes than these the proposed methodology gives, highly decrease performance. Regard-

European Union
European Social Fund
OPERATIONAL PROGRAMME
EDUCATION AND LIFELONG LEARNING
investing in knowledge society
MINISTRY OF EDUCATION, LIFELONG LEARNING AND RELIGIOUS AFFAIRS
MANAGING AUTHORITY
Co- financed by Greece and the European Union
NSRF
2007-2013
programmes for development
EUROPEAN SOCIAL FUND

ing shared caches, performance is not highly affected by the tiles selection, suffice tile sizes are not larger than the shared cache.

5 Conclusions

In this paper, a new methodology for Matrix-Matrix Multiplication using SIMD unit, at one and more cores having a shared cache, is presented. This is the first time for MMM, that i) the optimization is done by exploiting the major s/w and h/w parameters as one problem and not separately, ii) the memory hierarchy architecture details (e.g. data cache associativity) are fully exploited for this algorithm, iii) the final schedule is found by searching an orders of magnitude smaller exploration space.

Acknowledgements This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program ”Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF) - Research Funding Program: Heracleitus II. Investing in knowledge society through the European Social Fund.

References

1. Openblas, an optimized blas library (2012). URL available at http://xianyi.github.com/OpenBLAS/
2. Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O’Boyle, M.F.P., Thomson, J., Toussaint, M., Williams, C.K.I.: Using machine learning to focus iterative optimization. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO ’06, pp. 295–305. IEEE Computer Society, Washington, DC, USA (2006). DOI 10.1109/CGO.2006.37. URL http://dx.doi.org/10.1109/CGO.2006.37
3. Bacon, D.F., Graham, S.L., Oliver, Sharp, J.: Compiler transformations for high-performance computing. ACM Computing Surveys **26**, 345–420 (1994)
4. Bilmes, J., Asanović, K., Chin, C., Demmel, J.: 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)
5. Bjørstad, P., Manne, F., Sørevik, T., Vajtersic, M.: Efficient matrix multiplication on simd computers. SIAM J. MATRIX ANAL. APPL **13**, 386–401 (1992)
6. Blackford, L.S., Choi, J., Cleary, A., D’Azeuedo, E., Demmel, J., Dhillon, I., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK user’s guide. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1997)
7. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. SIGPLAN Not. **30**(8), 207–216 (1995). DOI 10.1145/209937.209958. URL http://doi.acm.org/10.1145/209937.209958

8. Chatterjee, S., Lebeck, A.R., Patnala, P.K., Thottethodi, M.: Recursive array layouts and fast parallel matrix multiplication. In: In Proceedings of Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 222–231 (1999)
9. Choi, J.: A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. p. 224 (1997)
10. Cooper, K.D., Subramanian, D., Torczon, L.: Adaptive optimizing compilers for the 21st century. Journal of Supercomputing **23**, 2002 (2001)
11. Desprez, F., Suter, F.: 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
12. Desprez, F., Suter, F.: Impact of mixed-parallelism on parallel implementations of the strassen and winograd matrix multiplication algorithms: Research articles. Concurr. Comput. : Pract. Exper. **16**(8), 771–797 (2004). DOI 10.1002/cpe.v16:8. URL http://dx.doi.org/10.1002/cpe.v16:8
13. Frigo, M., Johnson, S.G.: The fastest fourier transform in the west. Tech. rep., Cambridge, MA, USA (1997)
14. Garcia, E., Venetis, I.E., Khan, R., Gao, G.R.: 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, pp. 316–327. Springer-Verlag, Berlin, Heidelberg (2010). URL http://dl.acm.org/citation.cfm?id=1885276.1885308
15. Geijn, R.A.V.D., Watts, J.: Summa: Scalable universal matrix multiplication algorithm. Tech. rep. (1997)
16. Goto, K., van de Geijn, R.: On reducing tlb misses in matrix multiplication. Tech. rep. (2002)
17. Goto, K., van de Geijn, R.A.: Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. **34**(3), 12:1–12:25 (2008). DOI 10.1145/1356052.1356053. URL http://doi.acm.org/10.1145/1356052.1356053
18. Granston, E., Holler, A.: Automatic recommendation of compiler options. In: In Proceedings of the Workshop on Feedback-Directed and Dynamic Optimization (FDDO (2001)
19. Guennebaud, G., Jacob, B., et al.: Eigen v3. http://eigen.tuxfamily.org (2010)
20. Hall, J.D., Carr, N.A., Hart, J.C.: Cache and bandwidth aware matrix multiplication on the gpu. Tech. rep. (2003)
21. Hattori, M., Ito, N., Chen, W., Wada, K.: Parallel matrix-multiplication algorithm for distributed parallel computers. Syst. Comput. Japan **36**(4), 48–59 (2005). DOI 10.1002/scj.v36:4. URL http://dx.doi.org/10.1002/scj.v36:4
22. Hunold, S., Rauber, T.: Automatic tuning of pdgemm towards optimal performance. In: Proceedings of the 11th international Euro-Par conference on Parallel Processing, Euro-Par'05, pp. 837–846. Springer-Verlag, Berlin, Heidelberg (2005). DOI 10.1007/11549468_91
23. Hunold, S., Rauber, T., Rünger, G.: Multilevel hierarchical matrix multiplication on clusters. In: Proceedings of the 18th annual international conference on Supercomputing, ICS '04, pp. 136–145. ACM, New York, NY, USA (2004). DOI 10.1145/1006209.1006230. URL http://doi.acm.org/10.1145/1006209.1006230
24. Intel: Intel mkl, available at http://software.intel.com/en-us/intel-mkl (2012)
25. Jiang, C., Snir, M.: Automatic tuning matrix multiplication performance on graphics hardware. In: In the proceesings of the 14th International Conference on Parallel Architecture and Compilation Techniques (PACT), pp. 185–196 (2005)
26. Kisuki, T., Knijnenburg, P.M.W., O'Boyle, M.F.P., Bodin, F., Wijshoff, H.A.G.: A feasibility study in iterative compilation. In: Proceedings of the Second International Symposium on High Performance Computing, ISHPC '99, pp. 121–132. Springer-Verlag, London, UK, UK (1999). URL http://dl.acm.org/citation.cfm?id=646347.690219
27. Krishnan, M., Nieplocha, J.: Srumma: A matrix multiplication algorithm suitable for clusters and scalable shared memory systems. Parallel and Distributed Processing Symposium, International **1**, 70b (2004). DOI http://doi.ieeecomputersociety.org/10.1109/IPDPS.2004.1303000
28. Krishnan, M., Nieplocha, J.: Memory efficient parallel matrix multiplication operation for irregular problems. In: Proceedings of the 3rd conference on Computing frontiers,

CF '06, pp. 229–240. ACM, New York, NY, USA (2006). DOI 10.1145/1128022.1128054. URL http://doi.acm.org/10.1145/1128022.1128054

29. Krivutsenko, A.: Gotoblas - anatomy of a fast matrix multiplication. Tech. rep. (2008)
30. Kulkarni, M., Pingali, K.: An experimental study of self-optimizing dense linear algebra software. Proceedings of the IEEE **96**(5), 832–848 (2008)
31. Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J., Jones, D.: Fast searches for effective optimization phase sequences. SIGPLAN Not. **39**(6), 171–182 (2004). DOI 10.1145/996893.996863. URL http://doi.acm.org/10.1145/996893.996863
32. Kulkarni, P.A., Whalley, D.B., Tyson, G.S., Davidson, J.W.: Practical exhaustive optimization phase order exploration and evaluation. TACO **6**(1) (2009)
33. Kurzak, J., Alvaro, W., Dongarra, J.: Optimizing matrix multiplication for a short-vector simd architecture - cell processor. Parallel Comput. **35**(3), 138–150 (2009). DOI 10.1016/j.parco.2008.12.010. URL http://dx.doi.org/10.1016/j.parco.2008.12.010
34. Michaud, P.: 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, pp. 187–196. ACM, New York, NY, USA (2011). DOI 10.1145/1944862.1944890. URL http://doi.acm.org/10.1145/1944862.1944890
35. Milder, P.A., Franchetti, F., Hoe, J.C., Püschel, M.: Computer generation of hardware for linear digital signal processing transforms. ACM Transactions on Design Automation of Electronic Systems **17**(2) (2012)
36. Monsifrot, A., Bodin, F., Quiniou, R.: 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, pp. 41–50. Springer-Verlag, London, UK, UK (2002). URL http://dl.acm.org/citation.cfm?id=646053.677574
37. Moon, B., Jagadish, H.V., Faloutsos, C., Saltz, J.H.: Analysis of the clustering properties of the hilbert space-filling curve. IEEE Transactions on Knowledge and Data Engineering **13**, 2001 (2001)
38. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. **42**(6), 89–100 (2007). DOI 10.1145/1273442.1250746. URL http://doi.acm.org/10.1145/1273442.1250746
39. Nikolopoulos, D.S.: Code and data transformations for improving shared cache performance on smt processors. In: ISHPC, pp. 54–69 (2003)
40. Park, E., Kulkarni, S., Cavazos, J.: 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, pp. 65–74. ACM, New York, NY, USA (2011). DOI 10.1145/2038698.2038711. URL http://doi.acm.org/10.1145/2038698.2038711
41. Pinter, S.S.: Register allocation with instruction scheduling: A new approach (1996)
42. Rünger, G., Schwind, M.: Fast recursive matrix multiplication for multi-core architectures. Procedia Computer Science **1**(1), 67–76 (2010). International Conference on Computational Science 2010 (ICCS 2010)
43. See homepage for details: Atlas homepage (2012). Http://math-atlas.sourceforge.net/
44. Shobaki, G., Shawabkeh, M., Rmaileh, N.E.A.: Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach. ACM Trans. Archit. Code Optim. **10**(3), 14:1–14:31 (2008). DOI 10.1145/2512432. URL http://doi.acm.org/10.1145/2512432
45. Stephenson, M., Amarasinghe, S., Martin, M., O'Reilly, U.M.: Meta optimization: improving compiler heuristics with machine learning. SIGPLAN Not. **38**(5), 77–90 (2003). DOI 10.1145/780822.781141. URL http://doi.acm.org/10.1145/780822.781141
46. Strassen, V.: Gaussian elimination is not optimal. Numerische Mathematik **14**(3), 354–356 (1969)
47. Tartara, M., Crespi Reghizzi, S.: Continuous learning of compiler heuristics. ACM Trans. Archit. Code Optim. **9**(4), 46:1–46:25 (2013). DOI 10.1145/2400682.2400705. URL http://doi.acm.org/10.1145/2400682.2400705
48. Thottethodi, M., Chatterjee, S., Lebeck, A.R.: Tuning strassen's matrix multiplication for memory efficiency. In: In Proceedings of SC98 (CD-ROM (1998)

49. Triantafyllis, S., Vachharajani, M., Vachharajani, N., August, D.I.: Compiler optimization-space exploration. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, CGO '03, pp. 204–215. IEEE Computer Society, Washington, DC, USA (2003). URL http://dl.acm.org/citation.cfm?id=776261.776284
50. Tsilikas, G., Fleury, M.: Matrix multiplication performance on commodity shared-memory multiprocessors. In: Proceedings of the international conference on Parallel Computing in Electrical Engineering, PARELEC '04, pp. 13–18. IEEE Computer Society, Washington, DC, USA (2004). DOI 10.1109/PARELEC.2004.43. URL http://dx.doi.org/10.1109/PARELEC.2004.43
51. Whaley, R.C., Dongarra, J.: Automatically Tuned Linear Algebra Software. Tech. Rep. UT-CS-97-366, University of Tennessee (1997)
52. Whaley, R.C., Dongarra, J.: Automatically tuned linear algebra software. In: SuperComputing 1998: High Performance Networking and Computing (1998)
53. Whaley, R.C., Dongarra, J.: Automatically Tuned Linear Algebra Software. In: Ninth SIAM Conference on Parallel Processing for Scientific Computing (1999). CD-ROM Proceedings
54. Whaley, R.C., Petitet, A.: Minimizing development and maintenance costs in supporting persistently optimized BLAS. Software: Practice and Experience **35**(2), 101–121 (2005)
55. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project. Parallel Computing **27**(1–2), 3–35 (2001)
56. Yotov, K., Li, X., Ren, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P.: Is search really necessary to generate high-performance blas? Proceedings of the IEEE **93**(2) (2005)
57. Yuan, N., Zhou, Y., Tan, G., Zhang, J., Fan, D.: High performance matrix multiplication on many cores. In: Proceedings of the 15th International Euro-Par Conference on Parallel Processing, Euro-Par '09, pp. 948–959. Springer-Verlag, Berlin, Heidelberg (2009). DOI 10.1007/978-3-642-03869-3_87
58. Zhuravlev, S., Saez, J., Fedorova, A., Prieto, M.: Survey of scheduling techniques for addressing shared resources in multicore processors. ACM Computing Surveys (In Press)