J Supercomput (2015) 71:2644–2667
DOI 10.1007/s11227-015-1409-9

A methodology for speeding up matrix vector multiplication for single/multi-core architectures

Vasilios Kelefouras[1] · Angeliki Kritikakou[2] · Elissavet Papadima[1] · Costas Goutis[1]

Published online: 29 March 2015

Abstract In this paper, a new methodology for computing the Dense Matrix Vector Multiplication, for both embedded (processors without SIMD unit) and general purpose processors (single and multi-core processors, with SIMD unit), is presented. This methodology achieves higher execution speed than ATLAS state-of-the-art library (speedup from 1.2 up to 1.45). This is achieved by fully exploiting the combination of the software (e.g., data reuse) and hardware parameters (e.g., data cache associativity) which are considered simultaneously as one problem and not separately, giving a smaller search space and high-quality solutions. The proposed methodology produces a different schedule for different values of the (i) number of the levels of data cache; (ii) data cache sizes; (iii) data cache associativities; (iv) data cache and main memory latencies; (v) data array layout of the matrix and (vi) number of cores.

Keywords Matrix vector multiplication · Data cache · Cache associativity · Multi-core · SIMD · Memory management · Data reuse

1 Introduction

The Dense Matrix Vector Multiplication (MVM) is an important kernel in most varied domains and application areas. It constitutes the kernel operation for computations of signal and image processing, computer science, natural science, engineering, economics and everyday graphical visualizations. MVM software kernel is run

✉ Vasilios Kelefouras
kelefouras@ece.upatras.gr

1 Department of Electrical and Computer Engineering,University of Patras, Patras, Greece

2 Education and Research Department in Computer science and Electrical Engineering, University of Rennes 1-IRISA/INRIA, Rennes, France

everyday on a huge number of processors all over the world, from small embedded microcontrollers to big supercomputers. Hence, the need of a fast MVM software implementation arises. Moreover, by optimizing MVM software, we can substitute a big and expensive CPU with a smaller and cheaper, achieving the same performance. The complexity of MVM is $O(M \times N)$ and its execution time highly depends on the memory hierarchy utilization.

The separate optimization of the following sub-problems, i.e., the sub-problems of finding the schedules with the minimum numbers of (i) load/store instructions; (ii) addressing instructions; (iii) L1 data cache accesses; (iv) L2 data cache accesses and (v) main memory data accesses, cannot give an optimum solution in most cases; this is because the separate sub-problems optimization gives a different schedule for each sub-problem and these schedules cannot coexist (refining one degrades the other). In this paper, these sub-problems are optimized as one problem and not separately and the final schedule is found theoretically according to the memory hierarchy parameters. Furthermore, by utilizing the software (e.g., data reuse, data dependences) and hardware parameters (e.g., data cache size and associativity), the exploration space is decreased by orders of magnitude and therefore instead of searching the whole space, only a small number of solutions is tested.

Regarding multi-core architectures, we theoretically and experimentally show that MVM performance is restricted by main memory latency and bandwidth values and thus a high core utilization factor cannot be achieved.

The state-of-the-art (SOA) self/hand-tuning libraries for linear algebra and Fast Fourier Transform (FFT) algorithm, such as ATLAS [1], OpenBLAS [2], GotoBLAS2 [3], Eigen [4], Intel_MKL [5], PHiPAC [6], FFTW [7], and SPIRAL [8], 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). This is because the development of a self-tuning library is a difficult and time-consuming task, since (i) 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, and (ii) the optimum parameters for two slightly different architectures normally are different. Such a case is MVM, which is a major kernel in linear algebra and also the topic of this paper.

The aforementioned SOA libraries optimize all the above parameters separately by using heuristics and empirical techniques. The independent optimization of the back end compiler phases (e.g., transformations, register allocation), leads to inefficient binary code due to dependencies among them. These dependencies require all phases be optimized together as one problem and not separately. Toward this, much research has been done, especially in the past, either to simultaneously optimize only two phases, e.g., register allocation and instruction scheduling (the problem is known to be NP-complete) [9,10] or to apply predictive heuristics [11,12]; nowadays compilers and related works, apply (i) iterative compilation techniques [13–16], (ii) machine learning compilation techniques to restrict the configurations' search space [17–20], or (iii) both iterative compilation and machine learning compilation techniques [21,22]. 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 and an initial training phase lasting for days or weeks. This fundamental problem, i.e., optimization of all compiler phases together, has not been solved, resulting in low-quality solutions. As it is well known, splitting a problem into parts and optimizing them separately does not lead to the optimum solution; in fact, it is far from the optimum. However, for this algorithm, the proposed methodology optimizes the major hardware (e.g., the size of each level of data cache) and software (e.g., data reuse) parameters simultaneously as one problem and not separately, leading to high-quality solutions.

The major contributions of this paper are: (i) the introduction of a new MVM methodology which is faster than the ATLAS SOA library; (ii) the optimization is done by fully exploiting the major software and hardware parameters as one problem and not separately, giving a smaller search space and high-quality solutions; (iii) for the first time the memory hierarchy architecture details (e.g., data cache associativity and latency) are fully exploited for this algorithm; (iv) the proposed methodology, due to the major contribution of number (ii) above 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 evaluated in desktop personal computers, i.e., Intel Pentium core 2 duo and i7, by using the SIMD (Single Instruction Multiple Data) unit; the proposed methodology is compared with the *cblas_sgemv* routine of ATLAS which runs on general purpose processors only. Also Valgrind [23] tool and SimpleScalar simulator [24] are used to measure the total number of instructions executed and the number of L1 and L2 data cache accesses and misses.

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, experimental results are presented, and Sect. 5 is dedicated to conclusions.

2 Related work

One of the state-of-the-art libraries for linear algebra is ATLAS [1,25–29]; ATLAS is an implementation of a new style of 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.

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 MVM algorithm, its techniques for supplying different implementations of kernel operations concerning memory management are empirical and hence it does not provide a 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. Although the proposed methodology is written in C language using SSE intrinsics, it achieves speedup of 1.3 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 the use of assembly is beyond the scope of this paper).

To our knowledge, there are only a few research works in optimizing the dense MVM software: [30–33]. [30,31,33] are MVM implementations on GPU architectures while [32] describes a parallel MVM algorithm. However, many works exist in optimizing the sparse MVM on cluster architectures and GPUs (sparse MVM achieves lower complexity), such as [34–36]. In contrast to the proposed methodology, the above works find the final MVM scheduling and the tile sizes by searching, since they do not exploit all the hardware and software constraints. However, if these constraints are fully exploited, the optimum solution can be found by testing only a small number of solutions; in this paper, tile sizes are given by inequalities which contain the cache sizes and cache associativities.

Regarding multi-core architectures, the OpenMP programming model is used, e.g., in [37], several data decomposition schemes using the OpenMP programming model are examined showing that both row-wise and block-wise decomposition achieve the best overall performance.

Regarding cluster architectures, the k-NUMA methodology [38] optimizes the dense MVM problem in clusters of symmetric multiprocessor (SMP) nodes, based on the partial development of optimal algorithms. Furthermore, [39] shows that the optimal performance can be achieved by combining distributed-memory and shared-memory programming models (MPI and OpenMP approaches, respectively) instead of MPI only. The importance of combining OpenMP with MPI is also indicated in [40], which presents the MultiCore SUPort (MCSup) OpenMP add-on library to co-locate threads with the memory they are using so as to avoid remote memory accesses.

The proposed methodology is compared only with ATLAS library because (i) it is well known that ATLAS is one of the state-of-the-art libraries; (ii) the above-related works are oriented for specific architectures only; (iii) they do not propose any method that can be used by us to make our methodology more efficient; (iv) they exploit only a few number of hardware and software parameters; (v) they exploit the hardware and software parameters separately and (vi) the number of main memory and data cache accesses of the proposed methodology is less; given that the memory management problem is the performance-critical parameter here, we can deduce that the proposed methodology achieves higher performance.

Finally, [41] gives some basic background knowledge on optimizing MVM, while [42–52] present how hardware and software can work on scalable multi-processor systems for matrix algorithms.

3 Proposed methodology

This paper presents a new methodology for computing the matrix vector multiplication (MVM) in one and more cores using a shared cache. This methodology achieves high execution speed by fully and simultaneously exploiting the combination of production–consumption, data reuse and MVM parallelism, under the hardware constraints of (i) the number of cores; (ii) the number of levels of data cache hierarchy; (iii) the size of each level of data cache hierarchy; (iv) the number of CPU registers and the size and the number of SIMD registers (XMM/YMM); (v) associativities of the data caches; (vi) latencies of the data cache memories, and (vii) SSE instruction latencies. For different hardware parameters, different schedules for MVM are produced.

We define the MVM problem as $Y = Y + A \times X$, where A is of size $M \times N$.

MVM performance depends on the time needed for the (i) data to be loaded/stored; (ii) matrix operations to be executed; (iii) addressing instructions to be executed, and (iv) instructions to be loaded from L1 instruction cache. The time needed for the instructions to be fetched from L1 instruction cache, is lower than the above values, since the MVM code size is small and it always fits in L1 instruction cache. Furthermore, most of today's general purpose processors contain separate L1 data and instruction caches and thus we can assume that shared/unified L2/L3 caches, contain only data. This is because the MVM code size is small and fits in L1 instruction cache here.

Equations 1 and 2 approximate MVM execution time; Eq. 1 holds for architectures that matrix operations and addressing instructions are executed in parallel and Eq. 2 holds for architectures that do not.

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

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

The time needed to execute the matrix operations is given by the following two equations. Equation 3 is used if there is a separate multiplication unit working in parallel and Eq. 4 otherwise (the number of multiplications is larger than the number of additions).

$$T_{\text{matrix-operations}} = \text{Mul}_{\text{lat}} \times (N \times M) \tag{3}$$

$$T_{\text{matrix-operations}} = \text{Mul}_{\text{lat}} \times (N \times M) + \text{Add}_{\text{lat}} \times (N \times M - M), \tag{4}$$

where Mul_{lat} and Add_{lat} are the latencies of the multiplication and addition units, respectively.

The $T_{\text{addressing}}$ value is not a constant number; its value increases when (i) the number of levels of tiling increases, (ii) the tile sizes decrease, (iii) less elements are assigned into registers, and (iv) the loop unroll factor decreases.

Furthermore, $T_{\text{data}} \succ T_{\text{matrix-operations}}$ in most cases, i.e., if the data do not fit in L1 data cache, $T_{\text{data}} \succ T_{\text{matrix-operations}}$, since at the minimum $(N \times M + M + N)$ elements are accessed from the slower upper memory (arrays A, Y and X, respectively). Also, in most cases, $T_{\text{data}} \succ T_{\text{addressing}}$, i.e., if the data do not fit in L1 data cache. However, $T_{\text{matrix-operations}}$ value is a constant number, but T_{data} and $T_{\text{addressing}}$ are not;

by decreasing the T_{data} value, e.g., by applying loop tiling, the $T_{\text{addressing}}$ value is increased, while the number of the matrix operations remains constant.

To summarize, MVM performance depends on the T_{data} and $T_{\text{addressing}}$ values; given that (i) T_{data} and $T_{\text{addressing}}$ are dependent; (ii) the load/store unit and the execution unit, work in parallel, high performance is achieved, only for both low T_{data} and $T_{\text{addressing}}$ values. The separate optimization of the T_{data} and $T_{\text{addressing}}$ values, gives different schedules which cannot coexist (refining one degrades the other).

T_{data} value is found theoretically according to the memory hierarchy parameters but $T_{\text{addressing}}$ is not. This is because the number of addressing instructions highly depends on the target compiler and on the unroll factor values; thus, although it is well known that the number of addressing instructions is increased by (a) increasing the number of levels of tiling, (b) decreasing the tile sizes, (c) assigning less elements into registers and (d) decreasing the loop unroll factor, we cannot make a good approximation for the number of addressing instructions. This is why a small number of possible solutions has to be tested. However, we do not test all possible solutions to find the most efficient one, but a number that is orders of magnitude smaller. To find out the minimum T_{total} value according to Eqs. 1 and 2, only the schedules giving both low T_{data} and $T_{\text{addressing}}$ values are selected (these two values are dependent); since we cannot make a good approximation for the number of addressing instructions ($T_{\text{addressing}}$), the proposed methodology tests all solutions giving a low T_{data} value. T_{data} is found theoretically according to the memory hierarchy parameters.

To sum up, MVM performance depends on the $\max(T_{\text{data}}, T_{\text{addressing}})$ value. The proposed methodology minimizes this value by proposing a different schedule for different types of cores, number of cores, memory architecture parameters and data array layouts. The reminder of this paper presents all these schedules. The proposed methodology using one CPU core and all CPU cores, is given in Sects. 3.1 and 3.2, respectively.

3.1 Single-core CPUs

MVM performance highly depends on the data array layout of A in main memory, as matrix A is by far larger than X and Y arrays ($N \times M$ elements are accessed for matrix A while $N \times M + N + M$ elements are accessed in total). The default data array layout depends on the target compiler, e.g., Fortran stores the array data column-wise in main memory, while C row-wise.

However, if the elements of A are written in main memory neither row-wise nor column-wise, but tile-wise, i.e., the elements are written with the exact order they are fetched from main memory according to the corresponding memory access pattern, then the performance is highly increased.

On the other hand, by changing the data array layout of A, an extra overhead is introduced and thus it is not performance efficient in most cases; the overhead is large, as the data of A are reloaded and rewritten from/to main memory. However, there are several cases where in changing the data array layout of A is efficient; these are: (i) if MVM input data are produced by the current application at run time; in this case, the initialization of A and the change of its layout, are made together, decreasing the overhead, (ii) if the data array layout is precomputed, (iii) if the matrix is multiplied many times by different vectors, and (iv) if the data array layout is computed by a custom hardware module.

The proposed methodology gives different schedules when the data array layout of A is changed, i.e., (i)–(iv) above, and not. Furthermore, it gives different schedules whether the processor contains an SIMD unit (general purpose processors) or not (embedded processors, microcontrollers). The proposed methodology when array A is written in main memory tile-wise and no SIMD unit exists, is given in Sect. 3.1.1. The proposed methodology when array A is written in main memory row-wise and no SIMD unit exists, is given in Sect. 3.1.2. The proposed methodology for SIMD processors, is given in Sect. 3.1.3.

3.1.1 Tile-wise data array layout of A: CPUs without SIMD

In this case, the optimum production–consumption (N intermediate results are produced and directly consumed to compute the $Y[0]$) of array Y and the sub-optimum data reuse of the array X have been found by splitting the arrays into tiles (Tile0) according to the register file size (in Eq. 5) (Fig. 1). The register file inequality is:

$$(\text{FP\_RF} \geq (k_0 + p + 1))\&(\text{Int\_RF} \geq \text{addr\_variables}), \quad (5)$$

where FP_RF is the number of floating point registers, Int_RF is the number of integer registers, addr_variables is the number of the addressing variables (e.g., iterators), k_0 is the size of Tile0 ($k_0 = 5$ in Fig. 1) and p is the number of instructions that can be executed in parallel. In practice $k_0 \geq p$.

k_0, p and 1 registers are allocated for Y, A and X arrays, respectively. The elements of matrix A have no data reuse and each element of A is fetched just once; thus, just one register is needed to store the A's elements; however, if p instructions can be executed

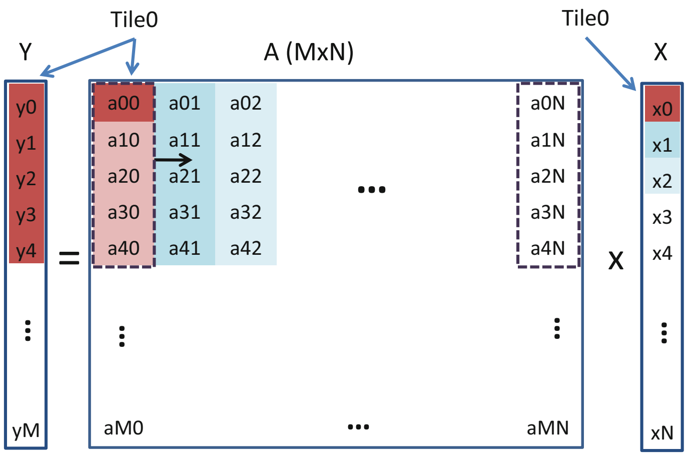


Fig. 1 The proposed methodology when $(k_0 + 1) \times N \prec L1$

in parallel, p registers are needed for matrix A to have all function units working in parallel ($p \geq 1$).

The scheduling follows. The first k_0 elements of the first column of A (Fig. 1) make a tile (Tile0). All the tile's elements are multiplied by the first-top element of X (data reuse of X) and their results are stored into the first-top k_0 elements of Y (Tile0). After the multiplication of all tile's elements by $X(0)$, the tile is moved by one position to the right and all its elements are multiplied by $X(1)$, etc. The results are always stored in-place, namely in the same registers, exploiting production–consumption of intermediate results of Y. According to the above schedule the (Y, X, A) arrays are fetched $(M, M^2/k0, M)$ instead of (M, M^2, M) times (when the row–column schedule is used), respectively.

Having only one level of a typical data cache, the time needed for the array elements to be loaded/stored, is approximated by Eq. 6:

$$T_{\text{data}} = \max\left(\frac{\text{L1r} \times \text{L1}_{\text{loads}} + \text{L1s} \times \text{L1}_{\text{stores}}}{\text{L1}_{\text{ports}}},\right.$$
$$\left.\frac{\text{MMr}}{\text{line}_{\text{L1}}} \times \text{MM}_{\text{loads}} + \text{MMs} \times \text{MM}_{\text{stores}} \times \frac{\lceil k_0/\text{line}_{\text{L1}} \rceil}{k_0}\right) \tag{6}$$

where L1r, L1s and MMr, MMs are the load/store latency values of L1 and main memory, respectively, and line_{L1} is the number of L1 data cache line elements. L1_{ports} is number of L1 read and write ports. L1_{loads}, $\text{L1}_{\text{stores}}$, MM_{loads} and $\text{MM}_{\text{stores}}$ are the number of elements loaded/stored from/to L1 and main memory, respectively. MMr value is not a constant number since it depends on the number of different main memory locations/pages are accessed. Regarding the number of main memory writes, $\lceil k_0/\text{line}_{\text{L1}} \rceil$ L1 data cache lines are written to main memory for each k_0 elements. For data cache architectures where reads and writes are executed in parallel, Eq. 6 is slightly different; we also assume that hardware prefetchers do not exist (which normally holds for embedded processors).

The L1_{loads}, $\text{L1}_{\text{stores}}$, MM_{loads} and $\text{MM}_{\text{stores}}$ values of Eq. 6 depend on the arrays size and on the data cache size. If the k_0 rows of A and X remain in L1 data cache, the number of L1 misses is low; on the other hand, if they are larger than the L1 data cache size, the number of the L1 misses increases, and therefore the number of L2/main memory accesses too.

In the case that the k_0 rows of A and the X fit in L1 data cache ($(k_0+1) \times N \times dtype \prec L1$), the T_{data} value of Eq. 6 is given by:

$$T_{\text{data}} = \max\left(\frac{\text{L1r}}{\text{L1}_{\text{ports}}} \times \left(M \times N + \frac{M \times N}{k_0}\right) + \frac{\text{L1s}}{\text{L1}_{\text{ports}}} \times M,\right.$$
$$\left.\frac{\text{MMr}}{\text{line}_{\text{L1}}} \times (M \times N + N) + \text{MMs} \times \frac{M \times \lceil k_0/\text{line}_{\text{L1}} \rceil}{k_0}\right) \tag{7}$$

The L1_{loads}, $\text{L1}_{\text{stores}}$, MM_{loads} and $\text{MM}_{\text{stores}}$ values of Eq. 7, are obtained from Table 1 (see 1 level of data cache and tiling for RF).

Table 1 Number of data accesses when A is of size $M \times M$

X does not fit in L1 $\left(\frac{\text{L1}\times(\text{assoc}-k)}{\text{assoc}} \prec k_0 \times M \times dtype\right)$			
1 level of data cache			
No optimization			
	Y	X	A
L1	M	M^2	M^2
DDR	M	M^2	M^2
Tiling for RF			
	Y	X	A
L1	M	M^2/k_0	M^2
DDR	M	M^2/k_0	M^2
Tiling for L1 and RF			
	Y	X	A
L1	M^2/k_1	M^2/k_0	M^2
DDR	M^2/k_1	M	M^2
X does not fit in L2 $\left(\frac{\text{L2}\times(\text{assoc}-k)}{\text{assoc}} \prec k_0 \times M \times dtype\right)$			
2 levels of data cache			
No optimization			
	Y	X	A
L1	M	M^2	M^2
L2	M	M^2	M^2
DDR	M	M^2	M^2
Tiling for RF			
	Y	X	A
L1	M	M^2/k_0	M^2
L2	M	M^2/k_0	M^2
DDR	M	M^2/k_0	M^2
Tiling for L1 and RF			
	Y	X	A
L1	M^2/k_1	M^2/k_0	M^2
L2	M^2/k_1	M	M^2
DDR	M^2/k_1	M	M^2
Tiling for L2, L1 and RF			
	Y	X	A
L1	M^2/k_1	M^2/k_0	M^2
L2	M^2/k_{2x}	M^2/k_{2y}	M^2
DDR	M^2/k_{2x}	M	M^2

Table 1 continued

X fits in L2 $\left(\frac{L2\times(\text{assoc}-k)}{\text{assoc}} \succ k_0 \times M \times dtype\right)$			
X does not fit in L1 $\left(\frac{\text{L1}\times(\text{assoc}-k)}{\text{assoc}} \prec k_0 \times M \times dtype\right)$			
2 levels of data cache			
Tiling for RF			
	Y	X	A
L1	M	M^2/k_0	M^2
L2	M	M^2/k_0	M^2
DDR	M	M	M^2
Tiling for L2, L1 and RF			
	Y	X	A
L1	M^2/k_1	M^2/k_0	M^2
L2	M^2/k_1	M^2/k_{2y}	M^2
DDR	M	M	M^2

On the other hand, in the case that the k_0 rows of A and the X do not fit in L1 data cache and for square matrix A of size $M \times M$, Eq. 6 and Table 1 give:

$$T_{\text{data}} = \max\left(\frac{\text{L1r}}{\text{L1}_{\text{ports}}} \times (M^2/k1 + M^2 + M^2/k_0) + \frac{\text{L1s}}{\text{L1}_{\text{ports}}} \times M^2/k_1,\right.$$
$$\left.\frac{\text{MMr}}{\text{line}_{\text{L1}}} \times (M^2/k1 + M^2 + M) + \text{MMs} \times M^2/k_1 \times \frac{\lceil k_0/\text{line}_{\text{L1}}\rceil}{k_0}\right) \quad (8)$$

To achieve the minimum T_{total} value (Eq. 1), all (k_0, k_1) values achieving small T_{data} values (Eq. 8) are selected (MMr $\succ$ L1r). Then these solutions are tested and the fastest is picked. These solutions are tested since we cannot approximate the $T_{\text{addressing}}$ value.

In the case that the sum of the k_0 rows of A and the X array size is larger than L1 data cache size, i.e., $(k_0 + 1) \times N \succ L1$, the X array does not fit in L1 data cache and thus it is not fetched once but N/k_0 times from the upper-slower memory of L1. To decrease the number of data accesses of the upper memory, loop tiling is applied by fully utilizing the L1 data cache size and associativity. Tiling is applied so the X array remains in L1 data cache since it is reused N/k_0 times.

Regarding the efficient use of L1 data cache, A and X arrays are partitioned into Tile1 tiles, of size $k0 \times k1$ and $k1$, respectively; the largest k_1 size is picked, for which the size of the data of one Tile1 of A (size of $k_0 \times k_1$) and one Tile1 of X (size of k_1) fit in L1 data cache, in Eq. 9.

$$\frac{\text{L1} \times (\text{assoc}_{\text{L1}} - k - 1)}{\text{assoc}_{\text{L1}}} \leq dtype \times k0 \times k1 \leq \frac{\text{L1} \times (\text{assoc}_{\text{L1}} - k)}{\text{assoc}_{\text{L1}}},$$
$$k = \lceil \frac{k1 \times dtype}{\text{L1}/\text{assoc}_{\text{L1}}} \rceil \leq \frac{\text{assoc}_{\text{L1}}}{2} \quad (9)$$

where $k1$ is the tile size in elements, $dtype$ is the size of the array elements in bytes (e.g., for float numbers $dtype = 4$), L1 is the size of L1 data cache, assoc_{L1} is L1 data cache associativity, $\text{assoc}_{\text{L1}} \succ 1$ here (e.g., for an 8-way set associative cache, $\text{assoc}_{\text{L1}} = 8$). k is an integer and it gives the number of L1 data cache lines with identical L1 addresses used for one Tile1 of X; for the remainder of this paper we will more freely say that we use k cache ways for X and $\text{assoc}_{\text{L1}} - k$ cache ways for A (in other words, A and X are written in separate data cache ways).

Let us explain Eq. 9, in more detail; Eq. 9 is introduced to minimize the number of L1 data cache misses and therefore the number of upper memory accesses. Given that each row of A is multiplied by X, X is reused M times, and thus, it has to remain in L1 data cache. To do this, cache lines of A are written in L1 without conflict with the X ones. This is achieved by loading k_0 sub-rows of A (Tile1 of A) and one sub-row of X (Tile1 of X), into separate L1 ways. This is achieved by storing the A array tile-wise in main memory (consecutive main memory locations) and by using ($k \times \frac{\text{L1}}{\text{assoc}_{\text{L1}}}$) L1 memory size for X and (($\text{assoc}_{\text{L1}} - k) \times \frac{\text{L1}}{\text{assoc}_{\text{L1}}}$) L1 memory size for A (in Eq. 9). We can more freely say that this is equivalent to using k cache ways for X and ($\text{assoc}_{\text{L1}} - k$) cache ways for A. An empty cache line is always granted (with respect to the size of the cache) for each different modulo of A and X memory addresses. It is important to say that if associativity is not taken into account and $\text{L1} \geq (k_0 \times k_1 + k_1)$ is used instead of in Eq. 9, the number of L1 misses will be much larger because A and X will conflict with each other. To our knowledge, this is the first time for an MVM algorithm that the cache associativity is utilized. Furthermore, the Y array is stored into main memory infrequently (usually 1–2 cache lines are written to memory when k_0 sub-rows of A have been multiplied by X); thus the number of L1 conflicts due to Y can be neglected (if a victim cache exists, then these conflicts never occur).

In the case that the data cache is direct mapped ($\text{assoc}_{\text{L1}} = 1$), loop tiling is not efficient since no tile can remain in data cache due to the cache modulo effect.

The scheduling of the Tile1 tiles follows (Fig. 2). All Tile1 of the first block column of A are multiplied by the first Tile1 of X. Then, all Tile1 of the second block column of A are multiplied by the second Tile1 of X; the procedure ends when all Tile1 of the last block column of A have been multiplied by the last Tile1 of X. Suppose an architecture with one level of data cache. Regarding L1 data cache, the numbers of load/store instructions for the (Y, A, X) arrays (A is a square matrix) are: $\text{L1}_{\text{loads}} = M \times (M/k_1, M, M/k0)$, $\text{L1}_{\text{stores}} = M \times (M/k_1)$. Regarding main memory (MM), the numbers of load/store instructions are the following: $\text{MM}_{\text{loads}} = M \times (M/k_1, M, 1)$, $\text{MM}_{\text{stores}} = M \times (M/k_1)$. If tiling for L1 data cache is not applied, the numbers of main memory load/store instructions are $M \times (1, M, M/k_0)$ and M, respectively (Table 1). Given that $k_1 \gg k0$, the number of main memory accesses is decreased.

The $k0$ value produced by in Eq. 5, is the value minimizing the number of load/store instructions and the number of register spills. Furthermore, the $k1$ value produced in Eq. 9 minimizes the number of L1 data cache misses (or equivalent number of main memory accesses here), given the $k0$ value. However, these two sub-problems depend on each other and therefore by decreasing the number of main memory data accesses, the number of L1 data accesses is increased. The number of the main memory data accesses is minimized when the $k1$ value is maximized, according to in Eq. 9;

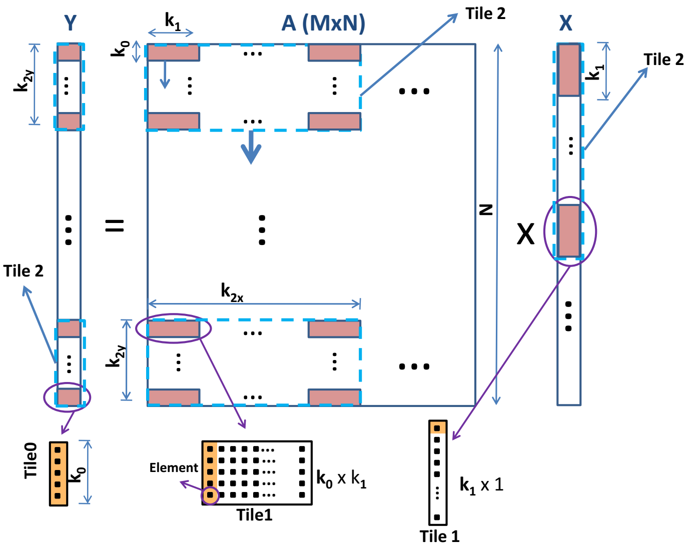


Fig. 2 The proposed methodology with three levels of tiling (L2, L1, RF). The tiles that are multiplied by each other have the *same color* (color figure online)

also the number of the L1 data cache accesses is minimized when the $k0$ value is maximized according to in Eq. 5; however, according to in Eq. 9, when k_1 increases the k_0 decreases (the maximum k_1 value is given for $k0 = 1$). There is a trade-off on the $(k0, k1)$ selection and thus the minimum T_{data} value depends on the target architecture parameters. The proposed methodology optimizes these two sub-problems together as one problem and not separately; the $(k0, k1)$ values are found theoretically according to the memory hierarchy architecture parameters, Eq. 6.

Furthermore, the Tile1 of A are chosen to be as large as possible in the x-axis only. This is because (i) the larger the tile of A, the larger the production and consumption of Y (as the Tile0 of A is moved by one position to the right in Fig. 1, the elements of Y are produced-consumed) (ii) by tiling only x-dimension, a lower number of addressing instructions is achieved (there is one loop less), and (iii) the larger the Tile1 of A is the larger the data reuse of X, as tiles are accessed in a column-wise way; this is because each Tile1 of X is fetched just once. Although the results are stored into memory N/k_1 times, the k_1 value is maximum, since Tile1 of A is as large as possible in the x-axis dimension.

Regarding 2-level data cache architectures, if the following inequality holds $M \times k_1 + M + k_1 \succ L2$ (the Y array, all Tile1 tiles of one block column of A and one Tile1 of X, do not fit in L2 data cache), the Y array is loaded/stored not once but M/k_1 times, from main memory. To decrease the number of main memory data accesses, loop tiling is applied by fully utilizing the L2 data cache size and associativity. Tiling is applied so that the Y array remains in L2 data cache, as it is loaded-reused N/k_1 times.

Regarding the efficient use of L2 data cache, the Y and A arrays are further split into Tile2 tiles (Fig. 2); one Tile2 of Y (size of k_{2y}), one Tile2 of A (size of $k_{2y} \times k_{k2x}$)

and one Tile2 of X size of k_{2x}, fit in L2 data cache, in Eq. 10 (Fig. 2). The Tile2 tiles are multiplied by each other exactly as the Tile1 tiles.

$$\frac{L2 \times (\text{assoc}_{\text{L2}} - 3)}{\text{assoc}_{\text{L2}}} \prec dtype \times k_{2y} \times k_{2x} \leq \frac{\text{L2} \times (\text{assoc}_{\text{L2}} - 2)}{\text{assoc}_{\text{L2}}} \tag{10}$$

In Eq. 10 satisfies that Tile2 of X and Tile2 of Y remain in L2 data cache; Tile2 of Y,A and X need 1, $\text{assoc}_{\text{L2}} - 2$ and 1, L2 ways, respectively; this means that the maximum L2 size needed for the Y, A and X arrays is $\text{L2}/\text{assoc}_{\text{L2}}$, $\frac{\text{L2}\times(\text{assoc}_{\text{L2}}-2)}{\text{assoc}_{\text{L2}}}$ and $\text{L2}/\text{assoc}_{\text{L2}}$, respectively.

Having these two tiles in L2, X is fetched just once while Y is fetched M/k_{2x} times, from main memory. The number of main memory accesses is minimized when $k_{2x} = M$, but in this case k_{2y} value is decreased and the number of L2 data cache accesses is increased (Table 1); there is trade-off and thus the k_{2x} and k_{2y} values are found theoretically according to the memory hit latency values. For most general purpose processors where L2 data cache size is large, $k_{2x} = M$. In most cases, we select $k_{2x} \gg k_{2y}$ since the main memory is many times slower than data cache.

The (k_{2y}, k_{2x}, k_1, k_0) values are computed by optimizing the following sub-problems, as one problem and not separately, as they depend on each other; these are the sub-problems of minimizing the number of L1, L2 and main memory accesses.

Having two levels of data cache, the time needed for the array elements to be fetched is:

$$\begin{aligned} T_{\text{data}} = \max\Bigg(& \frac{\text{L1r} \times \text{L1}_{\text{loads}} + \text{L1s} \times \text{L1}_{\text{stores}}}{\text{L1}_{\text{ports}}}, \\ & \frac{\text{L2r}}{\text{line}_{\text{L1}}} \times \text{L2}_{\text{loads}} + \text{L2s} \times \frac{\text{L2}_{\text{stores}} \times \lceil k_0/\text{line}_{\text{L1}} \rceil}{k_0}, \\ & \frac{MMr}{\text{line}_{\text{L2}}} \times \text{MM}_{\text{loads}} + \text{MMs} \times \frac{\text{MM}_{\text{stores}} \times \lceil k_0/\text{line}_{\text{L2}} \rceil}{k_0} \Bigg) \end{aligned} \tag{11}$$

where L2r and L2s are the L2 cache read and write latencies. In the case that $\text{modulo}(k_0, \text{line}_{\text{L2}}) \neq 0$ the number of main memory stores is slightly different as it depends on the $\text{modulo}(k_0, \text{line}_{\text{L2}})$ value.

Equation 11 and Table 1 give:

$$\begin{aligned} T_{\text{data}} = \max\Bigg(& \frac{\text{L1r} \times (M^2/k_1 + M^2/k_0 + M^2) + \text{L1s} \times M^2/k_1}{\text{L1}_{\text{ports}}}, \\ & \frac{\text{L2r}}{\text{line}_{\text{L1}}} \times (M^2/k_{2x} + M^2/k_{2y} + M^2) + \text{L2s} \times \frac{M^2/k_{2x} \times \lceil k_0/\text{line}_{\text{L1}} \rceil}{k_0}, \\ & \frac{\text{MMr}}{\text{line}_{\text{L2}}} \times (M^2/k_{2x} + M + M^2) + \text{MMs} \times \frac{M^2/k_{2x} \times \lceil k_0/\text{line}_{\text{L2}} \rceil}{k_0} \Bigg) \end{aligned} \tag{12}$$

Normally, $\text{MMr} \succ 40$, $3 \leq \text{L2r} \leq 7$ and $\text{L1r} = 1$. All $(k_0, k_1, k_{2y}, k_{2y})$ values giving a small T_{data} value according to the above inequalities are taken. Then, these schedules are tested and the fastest is picked. Furthermore, solutions for zero, one and two levels of tiling are selected, as tiling increases the number of load/store and addressing instructions.

Regarding full memory hierarchy utilization, one level of tiling for each level of data cache and one more for the register file is used; loop tiling decreases the number of data cache misses but increases the number of load/store and addressing instructions (typically one more level of tiling in MVM algorithm adds two additional loops). However, tiling for each level of data cache is not performance efficient in several cases, as the additional number of instructions (both load/store and arithmetic) may degrade performance.

To summarize, the T_{data} value for 1 level of tiling is given by Eqs. 7 and 8, while the T_{data} value or 2 levels of tiling is given by Eq. 12. Also, Eqs. 9 and 10, give the tile sizes.

3.1.2 Row-wise data array layout of A: CPUs without SIMD

Regarding register file utilization (if the k_0 rows of A and X fit in L1 data cache), the scheduling explained in the previous paragraph is used. However, in the row-wise case, if the register file size is large and thus k_0 value is large, the main memory bandwidth is increased, degrading performance; this is because the elements of each sub-row of A are written in different main memory locations (and also in different main memory pages); reading from a large number of different main memory locations needs more time. Therefore, k_0 value has an upper threshold which is bounded by main memory parameters (this is further explained into Sect. 3.2).

Regarding L1 data cache, in Eq. 9 does not hold in the row-wise case. If the data array layout of A is row-wise, the elements of the k_0 sub-rows of A are not written in consecutive main memory locations and thus they are not written in consecutive L1 data cache locations; actually for floating point values, they have $N \times 4$ bytes distance. This means that by using in Eq. 9, the Tile1 of X does remain in L1 data cache, as the cache lines of A will conflict with the cache lines of X. To overcome this problem, the following inequality is introduced.

$$k_0 \leq \mathrm{assoc}_{\mathrm{L1}} - 1 \tag{13}$$

According to in Eq. 13, one L1 way is used for each sub-row of A and another one for the Tile1 of X (assoc $\succ$ 1). Normally, the k_0 value of the row-wise case, is smaller than the k_0 value of the tile-wise case and, thus, a larger number of data accesses is achieved in this case. When in Eq. 13 holds, tiling is applied according to in Eq. 14; the maximum k_1 value is selected so that Tile1 of X fit in one L1 cache way.

$$dtype \times k_1 \leq \frac{\mathrm{L1}}{\mathrm{assoc}_{\mathrm{L1}}} \tag{14}$$

All the (k_0, k_1) values giving a small number of T_{data} according to the above inequalities are taken. To our knowledge, this is the first time that loop tiling is applied by analyzing (i) the data cache size, (ii) the data cache associativity and (iii) data arrays layout in main memory as one problem and not separately.

Regarding L2 data cache, tiling is applied as explained in the previous paragraph, but $k_{2x} = N$. In this case, all the sub-rows of each Tile2 of A are written in non-consecutive main memory locations and, thus, in non-consecutive data cache locations; therefore, the number of L2 data cache misses is increased. To minimize the number of L2 data cache misses, $k_{2x} = N$; in this case, all the sub-rows of Tile2 of A are of size N and they are written in consecutive main memory locations.

3.1.3 CPUs with SIMD

Most of the today's general purpose processors have Single Instruction Multiple Data (SIMD) unit in order to further increase performance by using vector operations. However, to efficiently use the SIMD unit, (i) the schedule decision must take into account the instruction selection problem, since there are instructions with variant latency/throughput values (e.g., 'hadd' instructions have very large latency and throughput values), and (ii) the data have to be aligned in main memory according to the SIMD registers size.

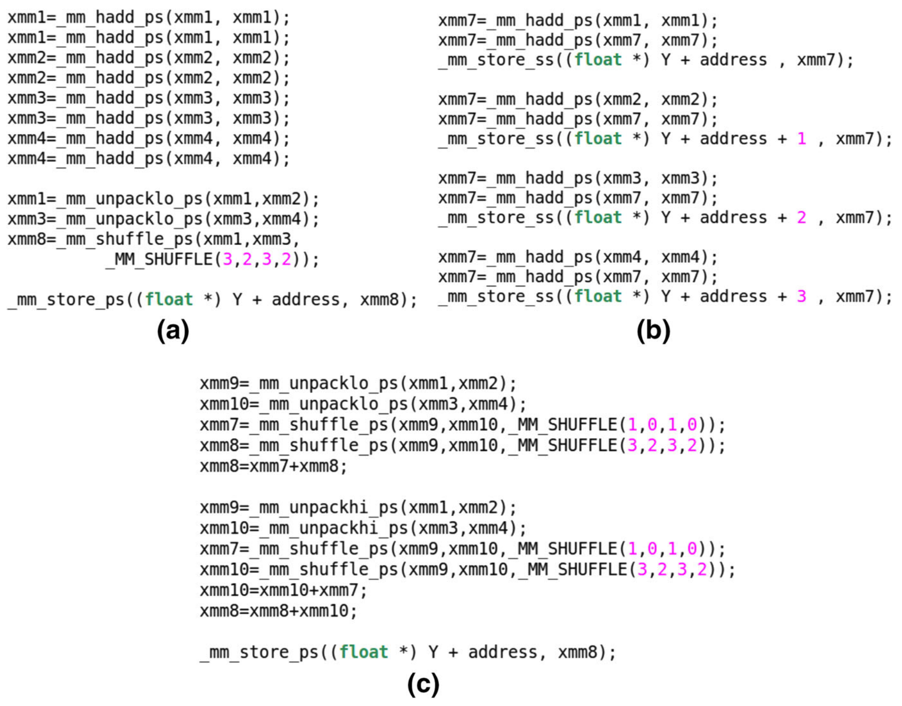
Regarding SIMD architectures, the schedule explained in Sects. 3.1.1 and 3.1.2, is not performance efficient. To implement the above schedule into AVX/SSE technology, several copies of each element of X have to be stored into 1 YMM/XMM register depending on the elements' size and registers' size; loading just one element needs more time than loading a whole L1 cache line and this is why this schedule is not performance efficient. SIMD unit introduces additional constraints which are fully exploited here.

The three arrays' data are stored into aligned main memory locations according to the SIMD register size; AVX technology supports 256-bit registers while SSE technology supports 128-bit registers. The optimum production–consumption of array Y and the sub-optimum data reuse of array X have been selected by splitting the arrays into tiles according to the number of XMM/YMM registers (Eq. 15).

$$\text{Regs} = k0 + 1 + 1, \tag{15}$$

where Regs is the number of the XMM/YMM registers and $k0$ is the number of the registers used for Y array. Thus, (Regs$-$2) registers for Y, 1 register for A and 1 register for X, are used.

The scheduling follows. Consider that there are 8 XMM registers (XMM0:XMM7) of size 128-bit and the arrays contain floating point data (4 bytes each). The first 4 elements of A and X are fetched into XMM0 and XMM1, respectively. They are multiplied by each other and the result is stored into XMM2. Then, the first 4 elements of the second row of A are loaded into XMM0 again; XMM0 is multiplied by XMM1 and the results is stored into XMM3. The procedure is repeated for the 3rd, 4th, 5th and 6th row of A (XMM2:XMM7 contain the values of $Y[0]$:$Y[5]$, respectively). Afterwards, the second quartet of X is loaded into XMM1 and the second quartet of the first row of A is loaded into XMM0; they are multiplied by each other and the result is accumulated into XMM2. Then the second quartet of the second row of A is loaded into XMM0 again, etc. When the first six rows of A have been fully multiplied by X, the four values of each one of XMM2:XMM7 registers are summed and the



```
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 *) Y + address, xmm8);
```

(a)

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

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

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

xmm7=_mm_hadd_ps(xmm4, xmm4);
xmm7=_mm_hadd_ps(xmm7, xmm7);
_mm_store_ss((float *) Y + 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 *) Y + address, xmm8);
```

(c)

Fig. 3 Three different ways for unpacking the multiplication results; XMM1, XMM2, XMM3, XMM4 contain the Y values. For most SIMD architectures, the three schedules are in increased performance order

results are stored into main memory (Y array), e.g., the sum of the four XMM2 values is $Y(0)$. The above procedure continues until all rows of A have been multiplied by X. The Y array is written in memory once, while array X is loaded $N/(\text{Regs} - 2)$ times.

There are several ways to sum the XMM2:XMM7 data: (i) to accumulate the values of each XMM register, to pack the results into new ones and to store four values in main memory at once (Fig. 3a); (ii) to accumulate the values of each XMM register and store each single Y value separately (Fig. 3b); (iii) to unpack the XMM values of the registers and to pack them into new ones in order to add elements of different registers (Fig. 3c). Within most architectures, the third one is faster than the other two, because unpacking and shuffle operations have smaller latency and throughput values than hadd operations. Also, the second is faster than the first within most architectures, because the store and add operations can be executed in parallel.

Regarding L1 data cache utilization, the schedule used is the same as in the previous subsections. However, loop-tiling is less effective here, because most of the today's general purpose processors have hardware prefetchers, for both L1 and L2 caches, decreasing the number of data cache misses.

By using SIMD, Eq. 2 changes into Eq. 16 as tiling further increases the number of SSE instructions; as the k_1 value decreases, the number of SSE instructions increases according to the following equation (code shown in Fig. 3 is executed more times).

$$T_{\text{data}} = \max\left(T_{\text{data}}, T_{\text{matrix-operations}} + c \times \frac{M}{k_1}, T_{\text{addressing}}\right), \tag{16}$$

where c is the sum of the SSE instruction latencies shown in Fig. 3.

3.2 Multi-core CPUs

Having more than one core, the MVM problem is partitioned into smaller MVM sub-problems. Each sub-problem corresponds to a thread and each thread is executed in one core only. Each thread must contain at least a specific number of instructions, to minimize the core 'idle-time'; this is because partitioning MVM into smaller sub-problems results in thread initialization and synchronization time comparable to the thread execution time, leading to low performance. This is an additional constraint.

Most of today's multi-core processors typically contain 2 or 3 levels of cache, having either (i) separate L1 data and instruction caches and a shared L2 cache or (ii) separate L1 data and instruction caches, separate unified L2 caches and a shared L3 cache, respectively. All caches have an LRU replacement policy.

The MVM problem is partitioned into threads according to the number of cores. All threads run in parallel, without communicating with each other. This is achieved by partitioning the matrix A horizontally into q equal parts and by partitioning the array Y into q equal parts, where q is the number of the cores. The multiplication of each part of A by X makes a different thread. The MVM problem is partitioned into q different threads and each thread writes on its own part of Y into main memory. Each thread is executed at only one core and the schedule is almost the same as in Sect. 3.1 (the differences are explained below).

By partitioning the MVM into threads and running them in parallel, the performance-critical parameters are main memory latency and bandwidth values. Although, data reuse and data cache hierarchy have been fully utilized, minimizing the number of X and Y main memory accesses, the elements of A (which are much more than X and Y), are used only once (no data reuse); thus, the number of main memory accesses of A cannot be reduced ($N \times M$ loads); for one matrix–vector multiplication to be executed, a different value of A is needed and this is why main memory latency cannot be hidden by the cache (CPU remains idle several cycles).

Thus, the MVM execution time depends on the time needed for matrix A to be loaded from main memory. This is why a large core utilization factor cannot be achieved. MVM performance is restricted by main memory latency and bandwidth values, so there is no way to increase performance more than this.

Furthermore, the main memory bandwidth would be low if we adopt the schedule explained in Sect. 3.1. This is because each thread accesses data from many different main memory locations. To be more precise, the elements of each sub-row of A are written in different main memory locations, so reading from different main memory locations needs more time. Therefore, as the k_0 value decreases, the time needed fetching the elements of A decreases. On the other hand, as the k_0 value increases according to Eq. 15, the number of load/store instructions and the number of data cache accesses decreases (however, it is not the critical parameter here). Therefore, the proposed methodology gives a different k_0 value according to the number of cores used and main memory parameters; as the number of the cores used increases, the k_0 value decreases.

In the case that the data array layout is tile-wise, MVM performance is highly increased. This is because (i) array A is accessed from consecutive main memory locations only, increasing the main memory bandwidth; (ii) the hardware prefetchers can prefetch the data of A because only one page is accessed; (iii) the X array fits in the private caches in most cases.

For commercial general purpose processors, the minimum number of main memory accesses is achieved even if no tiling for data cache is applied, since shared caches are very large. This is because: (i) the Y and A are accessed only once and (ii) the X array elements which are loaded many times, will remain in the shared cache (data reuse of X). The X array remains in shared cache because: (i) it is fetched many times by all the cores (LRU cache replacement policy), and (ii) its size is always smaller than one shared cache way size.

In the case that the cache hierarchy consists of 3 levels of cache and thus a private L2 cache for each core exists, the X array can fit in all the L2 memories even if no tiling for data cache is applied; in this way performance is increased even more. In the case that the data array layout of A is tile-wise, X always fits in L2 cache as L2 size in modern architectures is large enough. On the other hand, if the array layout of A is row-wise, in Eq. 17 must hold for the array X to remain in L2.

$$\frac{\text{L2}_{\text{private}} \times (\text{assoc} - k_0)}{\text{assoc}} \geq N \times dtype \tag{17}$$

where $\text{L2}_{\text{private}}$ is the size of the L2 private cache.

According to in Eq. 17, one L2 cache way is needed for each row of A. This is because the elements of the k_0 different rows of A are not in consecutive main memory locations and, thus, they are not fetched in consecutive L2 cache locations.

Regarding 3-level data cache architectures, tiling for data cache can be applied in several ways, i.e., tiling for (i) L1, (ii) L2 and (iii) both. However, by applying tiling for the cache, the number of main memory accesses is not further decreased, as both Y and X remain in L3 even if no tiling is applied because L3 shared cache size is very large in modern architectures; in general, performance is slightly affected by applying tiling for general purpose processors. This is because the critical parameter is the number of main memory accesses.

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 [53], a Pentium Intel Core 2 Duo T6600 at 2.20 GHz [54] and with a Pentium Intel i7-3930K (6 cores) at 3.2 GHz [55], all using SSE/AVX instructions and ATLAS 3.8.4, respectively. Also, the Valgrind [23] tool and SimpleScalar simulator [24] are used to measure the total number of L1 and L2 data cache accesses and misses. The first processor contains 8128-bit XMM registers,

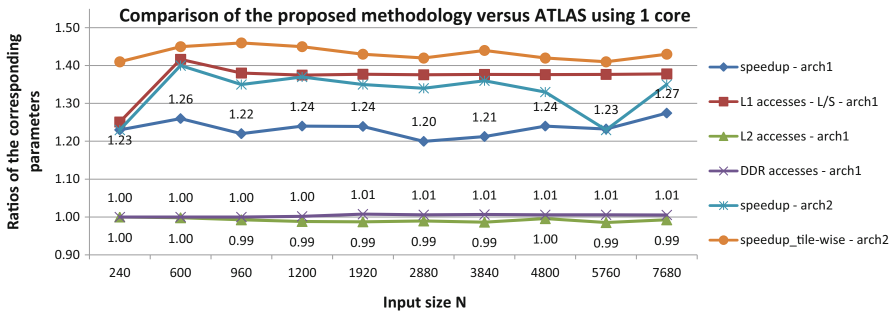


Fig. 4 Comparison of the proposed methodology over ATLAS on Pentium E6550 (arch1) and T6600 (arch2). The executables run in one core of the 2 that exist

L1 data and instruction caches of size 32 kbytes and shared L2 cache of size 4 Mbytes. The second processor contains 16 XMM registers, L1 data and instruction caches of size 32 kbytes and shared L2 cache of size 3 Mbytes. The third 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 12 Mbytes. All processors use the Operating system Ubuntu and the gcc-4.4.3 compiler. Furthermore, SimpleScalar simulator [24] is used to measure the number of data cache misses when no SIMD is used. In the experimental procedure, square matrix sizes ($N \times N$), including floating point numbers (4 bytes) as elements, were used. It is important to say that MVM thread has to be manually assigned to just one core; the programmer has to give 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 data movement involved.

First, a performance comparison is made for Intel E6550 (arch1) and Intel T6600 (arch2), by using only the one core of the 2 that exist (Fig. 4). The proposed methodology is compared with the *cblas_sgemv* routine of ATLAS and the average execution time among many executions is shown. Tiling for L1 and L2 cache is not performance efficient for Intel E6550 and Intel T6600. Regarding L2, tiling is not efficient because the whole Y and X arrays and several rows of A are of smaller size. Regarding L1, tiling is memory efficient but it is not performance efficient. Tiling is used in order to decrease the number of cache misses; however, the extra load/store and addressing instructions inserted here overlap the data locality advantage. Regarding register file utilization, both processors contain 8 XXM registers and thus loop tiling with $k_0 = 6$ is applied (in Eq. 15).

The proposed methodology is on average 1.23 and 1.33 faster than ATLAS library on E6550 (arch1) and T6600 (arch2), respectively; this is because it achieves a lower number of load/store instructions or equivalently L1 data cache accesses. For arch1, we achieve 1.38 times less load/store instructions than ATLAS. The number of load/store instructions is much less since the register file size is fully utilized (in Eq. 15); first, we use all the available XMM registers and second we use them efficiently according to the data reuse. Regarding the number of main memory data accesses, the proposed methodology achieves approximately the same number with ATLAS (we achieve about 1 % less main memory data accesses); the gain is small because L2 caches are

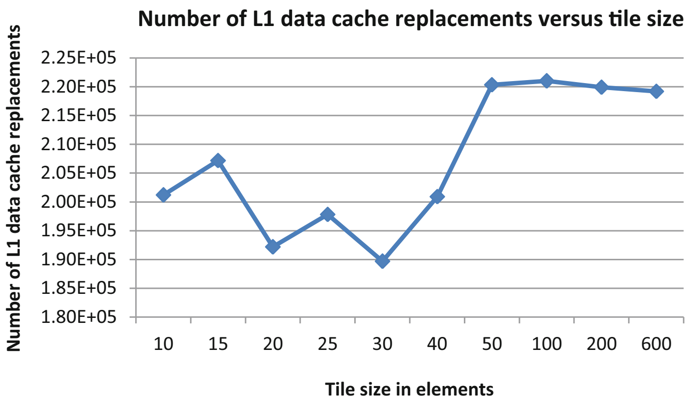


Fig. 5 Number of L1 data cache replacements for different tile sizes

large and the whole Y and X arrays and several rows of A can fit in L2. As it was explained in the previous subsections, the three arrays are accessed only once from main memory (Table 1) and thus the number of main memory data accesses is the minimum. Although we achieve only 1 % less main memory accesses, this gain is significant since only the number of Y and X data accesses can be decreased and not of A (matrix A is many times larger than Y and X); the $N \times M$ elements of A (in total $N \times M + N + M$) are accessed only once (no data reuse). Regarding the number of L2 cache accesses, the proposed methodology achieves approximately the same number with ATLAS (we achieve about 1 % more L2 data accesses). This is because ATLAS applies loop tiling for L1 here; it is important to say that we do not provide the schedule achieving the lower number of data cache accesses here, but the schedule achieving the best performance. Furthermore, the proposed methodology execution time increases proportionally to the matrix size, while ATLAS execution time does not; this is because ATLAS applies different schedules.

In the case that the data array layout of A is tile-wise, the speedup is further increased (about 1.42 %) for two reasons. First, the A elements are fetched from consecutive main memory locations (if the data array layout is row-wise, the elements of each sub-row of A are written from different main memory locations and reading from different memory addresses needs more time); in this way the main memory bandwidth is increased. Second, the hardware prefetchers can prefetch the data of A.

An evaluation for different tile sizes for L1 data cache is also performed on SimpleScalar simulator (Fig. 5). An architecture with one level of data cache of size 1 kbyte and 8-way associative is used; also, $N = 600$. According to in Eq. 14, the best tile size here is $T = 30$. This is verified by Fig. 5 too. As the tiles fit in L1 data cache, the number of cache replacements is low ($Tile \prec 30$), but when the tiles become larger than the L1 size, the number of L1 replacements becomes high ($Tile \succ 30$).

An evaluation for many cores of one CPU is also performed on i7-3930K (6 cores) (Fig. 6). To our knowledge, no related work exists for dense MVM on multi-core processors. Furthermore, the ATLAS library does not provide routines for more than one core. Thus, the speedups shown in Figs. 6 and 7 are over the proposed methodology for one core. By using only two cores, the speedup is (near)-optimum (from 1.88 up

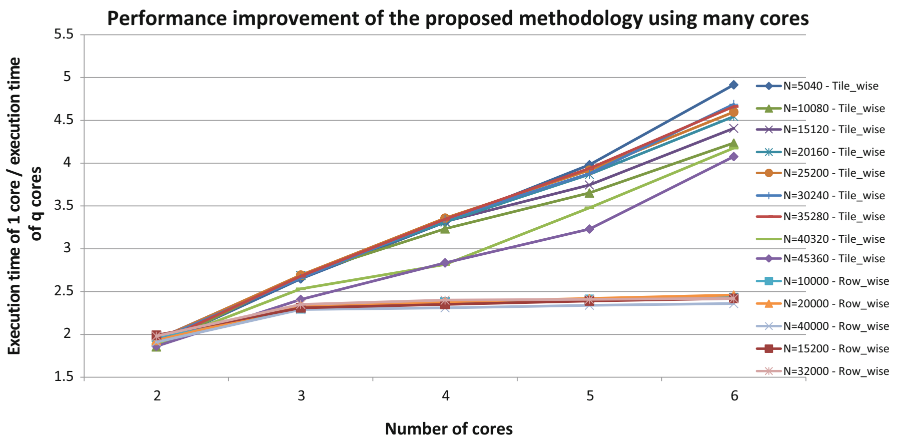


Fig. 6 Speedup of the proposed methodology for different number of cores

to 1.99, depending on the input size) for both row-wise and tile-wise case (Fig. 6). On the contrary, the gap between measured performance and optimum performance increases by using more than 2 cores; this is because performance is restricted by main memory latency and bandwidth values. Regarding row-wise data array layout of matrix A, performance is not increased by providing more than 4 cores since the main memory cannot provide the operands in lower time. The speedup is up to 2.45 (as the number of the cores increases, the proposed methodology gives a lower k_0 value as it has been explained in the proposed methodology subsection). Regarding tile-wise data array layout of matrix A, MVM performs much better (the time needed to change the data array layout of A is not included) as the performance gain increases proportionally to the number of cores. This is because: (i) all main memory accesses of A are performed at one memory page; (ii) the hardware prefetchers can prefetch the data of A as only one page is accessed, and (iii) the X array fits into the private caches in most cases. However, speedup cannot exceed 4.92 value. Regarding the two largest matrix sizes, i.e., $N = 45{,}360$ and $N = 40{,}320$, they achieve a slightly decreased performance, as the X array cannot remain in private L2 cache.

An evaluation for many different k_0 values is also performed on i7-3930K, for the row-wise case (Fig. 7). Intel i7-3930K contains 16,256-bit AVX registers; by using only one core ($q = 1$), only $k0 \geq 8$ values achieve high performance here. By using more than one core, the main memory bandwidth increases as the k_0 value decreases; this is because the elements of each sub-row of A are written in different main memory locations; reading from different addresses needs more time. On the other hand, as the k_0 value increases according to Eq. 15, the number of load/store instructions and the number of data cache accesses decreases. However, main memory is by far the most critical parameter here and, thus, a small $k0$ value is selected according to the main memory parameters. In the case that there are 5 or 6 cores, the main memory load is very high, and thus $k_0 = 1$ and $k_0 = 2$ are the best values; in this case, the register file and data cache utilization do not increase performance, as main memory is the critical parameter. For the same reason, $k0 = 1$ value achieves the best performance for $q = 6$ and the worst for $q = 1$.

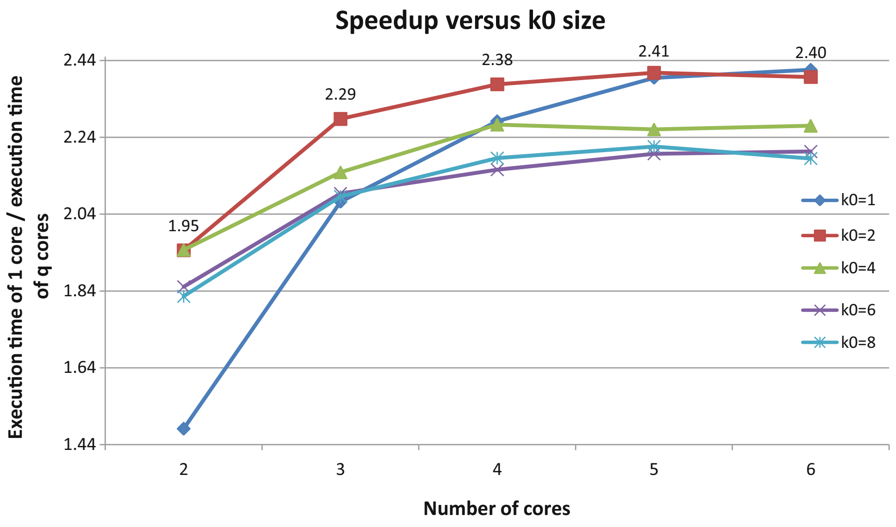


Fig. 7 Speedup of the proposed methodology for different k_0 values, $N = 20{,}160$

5 Conclusions

In this paper, a new methodology for computing the Dense Matrix Vector Multiplication for both embedded (single core processors without SIMD unit) and general purpose processors (single and multi-core processors, with SIMD unit) is presented. This methodology achieves higher execution speed than the ATLAS state-of-the-art library and the final schedule is found theoretically according to the memory hierarchy parameters. Furthermore, by utilizing software and hardware parameters, the number of solutions tested is orders of magnitude smaller; therefore, instead of searching the whole space, only a small number of solutions is tested.

References

1. Whaley RC, Petitet A (2005) Minimizing development and maintenance costs in supporting persistently optimized BLAS. Softw: Pract Exp 35(2):101–121
2. OpenBlas (2012). http://xianyi.github.com/OpenBLAS
3. Krivutsenko A (2008) GotoBLAS—anatomy of a fast matrix multiplication. Technical report
4. Guennebaud G, Jacob B et al (2010) Eigen v3. http://eigen.tuxfamily.org
5. Intel: Intel MKL (2012). http://software.intel.com/en-us/intel-mkl
6. Bilmes J, Asanović K, Chin C, Demmel J (1997) 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
7. Frigo M, Johnson SG (1997) The fastest Fourier transform in the west. Technical report. Cambridge, MA, USA
8. Milder P, Franchetti F, Hoe JC, Püschel M (2012) Computer generation of hardware for linear digital signal processing transforms. ACM Trans Des Autom Electron Syst 17(2) 15:1–15:33. doi:10.1145/2159542.2159547
9. Pinter SS (1996) Register allocation with instruction scheduling: a new approach. J Prog Lang 4(1):21–38

10. Shobaki G, Shawabkeh M, Rmaileh NEA (2008) Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach. ACM Trans Archit Code Optim 10(3):14:1–14:31. doi:10.1145/2512432
11. Bacon DF, Graham SL, Sharp OJ (1994) Compiler transformations for high-performance computing. ACM Comput Surv 26(4):345–420. doi:10.1145/197405.197406
12. Granston E, Holler A (2001) Automatic recommendation of compiler options. In: Proceedings of the workshop on feedback-directed and dynamic optimization (FDDO)
13. Triantafyllis S, Vachharajani M, Vachharajani N, August DI (2003) 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. http://dl.acm.org/citation.cfm?id=776261.776284
14. Cooper KD, Subramanian D, Torczon L (2002) Adaptive optimizing compilers for the 21st century. J Supercomput 23(1):7–22. doi:10.1023/A:1015729001611
15. Kisuki T, Knijnenburg PMW, O'Boyle MFP, Bodin F, Wijshoff HAG (1999) A feasibility study in iterative compilation. In: Proceedings of the 2nd international symposium on high performance computing, ISHPC '99, pp 121–132. Springer-Verlag, London, UK. http://dl.acm.org/citation.cfm?id=646347.690219
16. Kulkarni PA, Whalley DB, Tyson GS, Davidson JW (2009) Practical exhaustive optimization phase order exploration and evaluation. ACM Trans Archit Code Optim 6(1):1:1–1:36. doi:10.1145/1509864.1509865
17. Kulkarni P, Hines S, Hiser J, Whalley D, Davidson J, Jones D (2004) Fast searches for effective optimization phase sequences. SIGPLAN Not 39(6):171–182. doi:10.1145/996893.996863
18. Park E, Kulkarni S, Cavazos J (2011) 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. doi:10.1145/2038698.2038711
19. Monsifrot A, Bodin F, Quiniou R (2002) 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. http://dl.acm.org/citation.cfm?id=646053.677574
20. Stephenson M, Amarasinghe S, Martin M, O'Reilly UM (2003) Meta optimization: improving compiler heuristics with machine learning. SIGPLAN Not 38(5):77–90 (2003). doi:10.1145/780822.781141
21. Tartara M, Crespi Reghizzi S (2013) Continuous learning of compiler heuristics. ACM Trans Archit Code Optim 9(4):46:1–46:25. doi:10.1145/2400682.2400705
22. Agakov F, Bonilla E, Cavazos J, Franke B, Fursin G, O'Boyle MFP, Thomson J, Toussaint M, Williams CKI (2006) 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. doi:10.1109/CGO.2006.37
23. Nethercote N, Seward J (2007) Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not 42(6):89–100. doi:10.1145/1273442.1250746
24. Simplescalar CI, Burger D, Austin TM (1997) The SimpleScalar tool set, version 2.0. Technical report
25. Whaley RC, Petitet A, Dongarra JJ (2001) Automated empirical optimization of software and the ATLAS project. Parallel Comput 27(1–2):3–35
26. Whaley RC, Dongarra J (1999) Automatically tuned linear algebra software. In: 9th SIAM conference on parallel processing for scientific computing. CD-ROM Proceedings
27. Whaley RC, Dongarra J (1998) Automatically tuned linear algebra software. In: SuperComputing 1998: high performance networking and computing
28. Whaley RC, Dongarra J (1997) Automatically tuned linear algebra software. Technical report. UT-CS-97-366, University of Tennessee
29. See homepage for details: ATLAS homepage (2012). http://math-atlas.sourceforge.net/
30. Fujimoto N (2008) Dense matrix–vector multiplication on the CUDA architecture. Parallel Process Lett 18(4):511–530
31. Fujimoto N (2008) Faster matrix–vector multiplication on GeForce 8800GTX. In: IPDPS, pp 1–8. IEEE. http://dblp.uni-trier.de/db/conf/ipps/ipdps2008.html
32. Hendrickson B, Leland R, Plimpton S (1995) An efficient parallel algorithm for matrix–vector multiplication. Int J High Speed Comput 7:73–88

33. Sørensen HHB (2012) High-performance matrix–vector multiplication on the GPU. In: Proceedings of the 2011 international conference on parallel processing, Euro-Par'11, pp 377–386. Springer-Verlag, Berlin, Heidelberg
34. Zhang N (2012) A novel parallel scan for multicore processors and its application in sparse matrix–vector multiplication. IEEE Trans Parallel Distrib Syst 23(3):397–404. doi:10.1109/TPDS.2011.174
35. Williams S, Oliker L, Vuduc R, Shalf J, Yelick K, Demmel J (2007) Optimization of sparse matrix–vector multiplication on emerging multicore platforms. In: Proceedings of the 2007 ACM/IEEE conference on supercomputing, SC '07, pp 38:1–38:12. ACM, New York, NY, USA. doi:10.1145/1362622.1362674
36. Goumas G, Kourtis K, Anastopoulos N, Karakasis V, Koziris N (2009) Performance evaluation of the sparse matrix–vector multiplication on modern architectures. J Supercomput 50(1):36–77. doi:10.1007/s11227-008-0251-8
37. Michailidis PD, Margaritis KG (2010) Performance models for matrix computations on multicore processors using OpenMP. In: Proceedings of the 2010 international conference on parallel and distributed computing. Applications and Technologies, PDCAT '10, pp 375–380. IEEE Computer Society, Washington, DC, USA. doi:10.1109/PDCAT.2010.52
38. Schmollinger M, Kaufmann M (2002) Algorithms for SMP-clusters dense matrix–vector multiplication. In: Proceedings of the 16th international parallel and distributed processing Sysmposium, IPDPS '02, pp 57–. IEEE Computer Society, Washington, DC, USA. http://dl.acm.org/citation.cfm?id=645610.661893
39. Waghmare VN, Kendre SV, Chordiya SG (2011) Article: performance analysis of matrix–vector multiplication in hybrid (MPI + OpenMP). Int J Comput Appl 22(5):22–25. Published by Foundation of Computer Science
40. Baker AH, Schulz M, Yang UM (2011) On the performance of an algebraic multigrid solver on multicore clusters. In: Proceedings of the 9th international conference on high performance computing for computational science, VECPAR'10, pp 102–115. Springer-Verlag, Berlin, Heidelberg. http://dl.acm.org/citation.cfm?id=1964238.1964252
41. Parallel methods for matrix–vector multiplication. http://www.hpcc.unn.ru/mskurs/ENG/DOC/pp07.pdf
42. Bhandarkar SM, Arabnia HR (1995) The REFINE multiprocessor—theoretical properties and algorithms. Parallel Comput 21(11):1783–1805
43. Arabnia HR, Smith JW (1993) A reconfigurable interconnection network for imaging operations and its implementation using a multi-stage switching box. pp 349–357
44. Wani MA, Arabnia HR (2003) Parallel edge-region-based segmentation algorithm targeted at reconfigurable MultiRing network. J Supercomput 25(1):43–62
45. Arabnia HR (1990) A parallel algorithm for the arbitrary rotation of digitized images using process-and-data-decomposition approach. J Parallel Distrib Comput 10(2):188–192
46. Arabnia HR, Oliver MA (1989) A transputer network for fast operations on digitised images. Comput Graph Forum 8(1):3–11
47. Bhandarkar SM, Arabnia HR (1995) The Hough transform on a reconfigurable multi-ring network. J Parallel Distrib Comput 24(1):107–114
48. Arabnia HR, Oliver MA (1987) A transputer network for the arbitrary rotation of digitised images. Comput J 30(5):425–432
49. Arabnia HR, Bhandarkar SM (1996) Parallel stereocorrelation on a reconfigurable multi-ring network. J Supercomput 10(3):243–269
50. Arabnia HR, Oliver MA (1987) Arbitrary rotation of raster images with SIMD machine architectures. Comput Graph Forum 6(1):3–11
51. Bhandarkar SM, Arabnia HR, Smith JW (1995) A reconfigurable architecture for image processing and computer vision. Int J Pattern Recognit Artif Intell 9(02):201–229
52. Arabnia H (1995) A distributed stereocorrelation algorithm. In: Computer communications and networks, 1995. Proceedings, 4th international conference on, pp 479–482, IEEE
53. Intel core 2 duo processor E6550. http://ark.intel.com/Product.aspx?id=30783
54. Intel core 2 duo processor T6600. http://ark.intel.com/products/37255/Intel-Core2-Duo-Processor-T6600
55. Intel i7-2600K Processor. http://ark.intel.com/products/52214