Batched Cholesky Factorization for tiny matrices

Florian Lemaitre [1,2] Lionel Lacassagne [2]
[1] CERN on behalf of the LHCb Collaboration
[2] Sorbonne Universites, UPMC Univ Paris 06, CNRS UMR 7606, LIP6, Paris, France
florian.lemaitre@cern.ch – lionel.lacassagne@lip6.fr

***Abstract*—Many linear algebra libraries, such as the Intel MKL, Magma or Eigen, provide fast Cholesky factorization. These libraries are suited for big matrices but perform slowly on small ones. Even though State-of-the-Art studies begin to take an interest in small matrices, they usually feature a few hundreds rows. Fields like Computer Vision or High Energy Physics use tiny matrices. In this paper we show that it is possible to speedup the Cholesky factorization for tiny matrices by grouping them in batches and using highly specialized code. We provide High Level Transformations that accelerate the factorization for current Intel *SIMD* architectures (*SSE*, *AVX2*, *KNC*, *AVX512*). We achieve with these transformations combined with *SIMD* a speedup from 13 to 31 for the whole resolution compared to the naive code on a single core *AVX2* machine and a speedup from 15 to 33 with multithreading compared to the multithreaded naive code.**

I. Introduction

Linear algebra is everywhere, especially in scientific computation. There are a lot of fast linear algebra libraries like MKL, Magma or Eigen. However, these libraries are usually conceived with big matrices in mind. Experience shows that these libraries are not adapted for tiny matrices and perform slow on them.

Some fields like Computer Vision [14] or High Energy Physics often use tiny matrices, and require linear algebra different from classical High Performance Computing. Matrices up to a few dozen of rows are usual, for example through Kalman Filter [8] (in [4], they use a Kalman Filter with 4 dimensions). People begin to take an interest in small matrices [15], [17]. These studies are focused in matrices much larger than *SIMD* register size.

The goal of this paper is to present an optimized implementation of the Cholesky factorization for tiny matrices. It consists in a set of portable linear algebra routines and functions written in *C*. The chosen way to do it is to solve systems by batch, and parallelizing along matrices instead of parallelizing inside one single factorization. Our approach is similar to Spiral [16] or ATLAS [3]: we compare many different implementations of the same algorithm to keep the best one.

This paper presents the Cholesky factorization by batch. We will first expose the Cholesky algorithm. Then we will explain the transformations we made to improve the performance for tiny matrices. And finally, we will present the result of the benchmarks.

II. Cholesky Algorithm

The whole resolution is composed of 3 steps: the Cholesky factorization (aka decomposition), the forward substitution and the backward substitution. The substitution steps are grouped together.

A. Cholesky Factorization

The Cholesky factorization is a linear algebra algorithm used to express a symmetric positive-definite matrix as the product of a triangular matrix with its transposed matrix: $A = L \cdot L^T$ (algorithm 1).

The Cholesky factorization of a $n \times n$ matrix has a complexity in term of floating point operation of $n^3/3$ that is half of the LU one ($2n^3/3$), and is numerically more stable [9], [10]. This algorithm is naturally in-place as every input element is accessed only once and before writing the associated element of the output: L and A can be the same storage.

TABLE I: Number of floating point operations

(a) Regular

Algorithm	flop	load + store	AI
factorize	$\frac{1}{6}\left(2n^3 + 3n^2 + 7n\right)$	$\frac{1}{6}\left(2n^3 + 16n\right)$	~ 1
substitute	$2n^2$	$2n^2 + 4n$	~ 1
substitute1	$2n^2$	$2n^2 + 4n$	~ 1
solve	$\frac{1}{6}\left(2n^3 + 15n^2 + 7n\right)$	$\frac{1}{6}\left(2n^3 + 12n^2 + 40n\right)$	~ 1

(b) Scalarized

Algorithm	flop	load + store	AI
factorize	$\frac{1}{6}\left(2n^3 + 3n^2 + 7n\right)$	$\frac{1}{2}\left(2n^2 + 5n\right)$	$\sim n/3$
substitute	$2n^2$	$\frac{1}{2}\left(n^2 + 5n\right)$	~ 4
substitute1	$2n^2$	n	$2n$
solve	$\frac{1}{6}\left(2n^3 + 15n^2 + 7n\right)$	$\frac{1}{2}\left(n^2 + 6n\right)$	$\sim 2n/3$

Algorithm 1: Cholesky Factorization

```
input  : $A$   // $n \times n$ symmetric positive-definite matrix
output : $L$   // $n \times n$ lower triangular matrix
1  for $j = 0 : n-1$ do
2      $s \leftarrow A(j,j)$
3      for $k = 0 : j-1$ do
4          $s \leftarrow s - L(j,k)^2$
5      $L_{j,j} \leftarrow \sqrt{s}$
6      for $i = j+1 : n-1$ do
7          $s \leftarrow A(i,j)$
8          for $k = 0 : j-1$ do
9              $s \leftarrow s - L(i,k) \cdot L(j,k)$
10         $L(i,j) \leftarrow s / L(j,j)$
```

B. Substitution

Once we have the factorized form of A, we are able to solve easily systems like: $A \cdot X = R$. Indeed, if $A = L \cdot L^T$, the equation is equivalent to $L \cdot L^T \cdot X = R$. Triangular systems are easy to solve using the substitution algorithm.

The equation can be written like this: $L \cdot Y = R$ with $Y = L^T \cdot X$. So we need to first solve $L \cdot Y = R$ (forward substitution) and then to solve $L^T \cdot X = Y$ (backward substitution). Those two steps are group together to entirely solve a Cholesky factorized system (algorithm 2). Like the factorization, substitutions are naturally in-place algorithms: R, Y and X can be the same storage.

Algorithm 2: Substitution

```
input  : $L$   // $n \times n$ lower triangular matrix
input  : $R$   // vector of size $n$
output : $X$   // vector of size $n$, solution of $L \cdot L^T \cdot X = R$
temp   : $Y$   // vector of size $n$
// Forward substitution
for $i = 0 : n-1$ do
    $s \leftarrow R(i)$
    for $j = 0 : i-1$ do
        $s \leftarrow s - L(i,j) \cdot Y(j)$
    $Y(i) \leftarrow s / L(i,i)$
// Backward substitution
for $i = n-1 : 0$ do
    $s \leftarrow Y(i)$
    for $j = i+1 : n-1$ do
        $s \leftarrow s - L(j,i) \cdot X(j)$
    $X(i) \leftarrow s / L(i,i)$
```

C. Batch

With small matrices, all the dimensions are small. But parallelization is not efficient on small dimensions. A 3-iteration loop can not be efficiently vectorized.

The idea is to add one extra and long dimension to compute the Cholesky factorization of a large number of matrices instead of one.

Our problem now has a long dimension which can be parallelized with both vectorization and multi-threading. The principle is to have a `for` loop iterating over the matrices, and within this loop, compute the factorization of the matrix. This is also the approach used in [5], [6].

III. Transformations

Improving the performance of software requires transformations of the code, and especially High Level Transforms. For Cholesky, we made the following transforms:

- memory layout transform [2],
- loop transforms (loop unwinding [13], loop unrolling and unroll&jam),
- Architectural transforms: *SIMD*ization + fast square root.

A. Memory Layout Transform

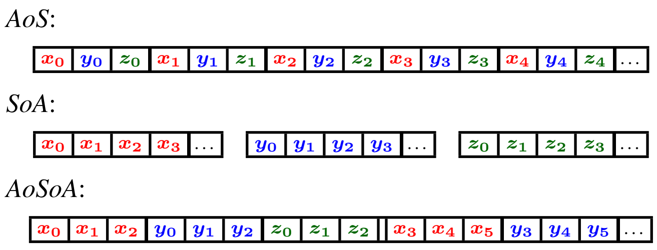
The memory layout is the first transformation considered. The most important aspect of the memory layout is the battle between *AoS* (Array of Structures) and *SoA* (Structure of arrays) [1] (Figure 1).

The *AoS* memory layout is the natural way to store arrays of objects in *C*. It consists in putting full objects one after the other. The code to access the `x` member of the i^{th} element of an array A looks like this: `A[i].x`. This memory layout uses only one active pointer and reduces the systematic cache eviction. The systematic cache eviction appears when multiple pointers share the same least significant bits and the cache associativity is not high enough to cache them all. But this memory layout is difficult to vectorize because the "`x`s" are not contiguous in memory.

The *SoA* memory layout addresses the vectorization problem. The idea is to have one array per members, and group them inside a structure. The access is written: `A.x[i]`. This memory layout is the default one in Fortran 77. It helps the vectorization of the code. But it uses as many active pointers as the number of members of the objects and can increase the number of systematic cache eviction when the number of active pointers is higher than the cache associativity.

The *AoSoA* memory layout (aka Hybrid *SoA*) tries to combine the advantages of *AoS* and *SoA*. The idea is to have a *SoA* memory layout of fixed size, and packing these structures into an array. Thus, it gives the same opportunity to vectorize as with *SoA*, but it keeps only one active pointer like in *AoS*. A typical value for the size of the *SoA* part is the *SIMD* register cardinal (or a small multiple of it). This access scheme can be simplified when iterating over such objects. The loop over the elements is split into two nested loops: one iterating over the *AoS* part, and one iterating over the *SoA* part. It is harder to write, especially to deal with boundaries.

The *SoA* memory layout was not used in this paper, and the term *SoA* will refer to the hybrid memory layout for the next part of this paper.



AoS:
x_0 y_0 z_0 x_1 y_1 z_1 x_2 y_2 z_2 x_3 y_3 z_3 x_4 y_4 z_4 ...
SoA:
x_0 x_1 x_2 x_3 ...
y_0 y_1 y_2 y_3 ...
z_0 z_1 z_2 z_3 ...
AoSoA:
x_0 x_1 x_2 y_0 y_1 y_2 z_0 z_1 z_2 x_3 x_4 x_5 y_3 y_4 y_5 ...

Fig. 1: Memory Layouts

The alignment of the data is also very important. The hardware has some requirements on the addresses of the elements. It is easier (if not mandatory) for the CPU to load a register from memory when the address is a multiple of the register size. In scalar code, `float` loads must be aligned with 4 bytes. This is done by the compiler automatically. However, vector registers are larger. The load address must be a multiple of the size of the *SIMD* register: 16 for *SSE*, 32 for *AVX* and 64 for *AVX512*. Aligned memory allocation should be enforced by specific functions like `posix_memalign` or `_mm_malloc`. One might also want to align data with the cache size (usually 64 bytes). This may improve cache hits by avoiding data being split into multiple cache lines when they fit within one cache line and avoid false sharing between threads.

The way data are stored and accessed is also important. The usual way to deal with multidimensional arrays in *C* is to linearize the addresses. For example, a $N\times M$ 2D array will be allocated like a 1D array with $N \cdot M$ elements. $A(i,j)$ is accessed with `A[i×M+j]`.

The knowledge of the actual size including the padding is required to access elements. Iliffe vectors [11] allow to access multi-dimensional arrays more easily. They consist of a 1D array plus an array of pointers to the rows. $A(i,j)$ is accessed through an Iliffe vector with `A[i][j]` (see Figure 2). It allows to easily store arrays of variable length rows like triangular matrices or padded/shifted arrays. It is easily extensible to higher dimensions.

With this memory layout, it is still possible to get the address of the data beginning, and use it like a linearized array. The allocation of an Iliffe vector needs extra space for the array of pointers. It also requires an initialization of the pointers before any use. As we work with pre-allocated arrays, the initialization of the pointers is not part of the benchmarks.

Accessing an Iliffe vector requires to dereference multiple pointers. It is possible to access the elements of an Iliffe vector like a linearized array. Keeping the last accessed position allows to avoid the computation of the new linearized address. Indeed, the new address can be obtained by moving the pointer to the previous address.

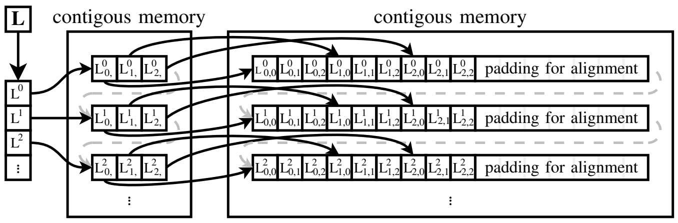


Fig. 2: Iliffe vector example: array of 3×3 matrices aligned with 64 bytes

B. Loop unwinding

It is possible to completely unroll the loops (see algorithms 3 and 4) as the matrices are tiny. This technique has several advantages:

- it avoids branching,
- it allows to keep all temporary results into registers (scalarization),
- it helps out-of-order processors to efficiently reschedule instructions,
- it can be combine with data prefetch.

When the factorization and the substitution are merged together, lines 18–21 of algorithm 3 and lines 2–5 of algorithm 4 disappear reducing the amount of memory accesses (Table Ib). This transform is very important as the algorithm is memory bound. One can see that the arithmetic intensity of the scalarized version is higher, and even higher when the steps are fused together leading to a version without intermediate memory access.

The register pressure is higher and the compiler may generate spill code to temporarily store variables into memory. It also makes the assembly code of the function bigger and may have a negative impact on the instruction cache.

Algorithm 3: Factorization unwinded for 4×4 matrices

```
input  : $A$   // $4\times4$ symmetric positive-definite matrix
output : $L$   // $4\times4$ lower triangular matrix
1  // Load $A$ into registers
2  $a_{00} \leftarrow A(0,0)$
3  $a_{10} \leftarrow A(1,0)$   $a_{11} \leftarrow A(1,1)$
4  $a_{20} \leftarrow A(2,0)$   $a_{21} \leftarrow A(2,1)$   $a_{22} \leftarrow A(2,2)$
5  $a_{30} \leftarrow A(3,0)$   $a_{31} \leftarrow A(3,1)$   $a_{32} \leftarrow A(3,2)$   $a_{33} \leftarrow A(3,3)$
6  // Factorize $A$
7  $l_{00} \leftarrow \sqrt{a_{00}}$
8  $l_{10} \leftarrow a_{10}/l_{00}$
9  $l_{20} \leftarrow a_{20}/l_{00}$
10 $l_{30} \leftarrow a_{30}/l_{00}$
11 $l_{11} \leftarrow \sqrt{a_{11} - {l_{10}}^2}$
12 $l_{21} \leftarrow \left(a_{21} - l_{20} \cdot l_{10}\right)/l_{11}$
13 $l_{31} \leftarrow \left(a_{31} - l_{30} \cdot l_{10}\right)/l_{11}$
14 $l_{22} \leftarrow \sqrt{a_{22} - {l_{20}}^2 - {l_{21}}^2}$
15 $l_{32} \leftarrow \left(a_{32} - l_{30} \cdot l_{20} - l_{31} \cdot l_{21}\right)/l_{22}$
16 $l_{33} \leftarrow \sqrt{a_{33} - {l_{30}}^2 - {l_{31}}^2 - {l_{32}}^2}$
17 // Store $L$ into memory
18 $L(0,0) \leftarrow l_{00}$
19 $L(1,0) \leftarrow l_{10}$   $L(1,1) \leftarrow l_{11}$
20 $L(2,0) \leftarrow l_{20}$   $L(2,1) \leftarrow l_{21}$   $L(2,2) \leftarrow l_{22}$
21 $L(3,0) \leftarrow l_{30}$   $L(3,1) \leftarrow l_{31}$   $L(3,2) \leftarrow l_{32}$   $L(3,3) \leftarrow l_{33}$
```

C. Loop Unroll & Jam

The Cholesky factorization of $n\times n$ matrices involves n square roots + n divisions for a total of $\sim n^3/3$ floating point operations (see Table I). The time before the execution of two data independent instructions (aka: throughput) is smaller than the latency. The latency of pipelined instructions can be hidden by executing in the pipeline another instruction without any data-dependence with the previous. The *ipc* (instructions per cycle) is then limited by the throughput of the instruction and not by its latency. Note that the "throughput" term used in the Intel documentation is the inverse of classical throughput: it is the number of cycles to wait between the launch of two consecutive instructions. If the throughput is less than 1, several instructions can be launched during the same cycle.

As current processors are Out-of-Order, they can reschedule instructions on-the-fly in order to execute in pipeline data-independent instructions. The size of the rescheduling window is limited and the processor may not be able to reorder instructions efficiently. In order to help the processor to pipeline instructions, it is possible to unroll loops and to interleave

Algorithm 4: Substitution unwinded for $4{\times}4$ matrices

```
input  : $L$   // $4{\times}4$ lower triangular matrix
input  : $R$   // vector of size $4$
output : $X$   // vector of size $4$, solution of $L \cdot L^T \cdot X = R$
1  // Load $L$ into registers
2  $l_{00} \leftarrow L(0,0)$
3  $l_{10} \leftarrow L(1,0)$   $l_{11} \leftarrow L(1,1)$
4  $l_{20} \leftarrow L(2,0)$   $l_{21} \leftarrow L(2,1)$   $l_{22} \leftarrow L(2,2)$
5  $l_{30} \leftarrow L(3,0)$   $l_{31} \leftarrow L(3,1)$   $l_{32} \leftarrow L(3,2)$   $l_{33} \leftarrow L(3,3)$
6  // Load $R$ into registers
7  $r_0 \leftarrow R(0)$
8  $r_1 \leftarrow R(1)$
9  $r_2 \leftarrow R(2)$
10 $r_3 \leftarrow R(3)$
11 // Forward substitution
12 $y_0 \leftarrow r_0/l_{00}$
13 $y_1 \leftarrow (r_1 - l_{10} \cdot y_0)/l_{11}$
14 $y_2 \leftarrow (r_2 - l_{20} \cdot y_0 - l_{21} \cdot y_1)/l_{22}$
15 $y_3 \leftarrow (r_3 - l_{30} \cdot y_0 - l_{31} \cdot y_1 - l_{32} \cdot y_1)/l_{33}$
16 // Backward substitution
17 $x_3 \leftarrow y_3/l_{33}$
18 $x_2 \leftarrow (y_2 - l_{32} \cdot x_3)/l_{22}$
19 $x_1 \leftarrow (y_1 - l_{21} \cdot x_2 - l_{31} \cdot x_3)/l_{11}$
20 $x_0 \leftarrow (y_0 - l_{10} \cdot x_1 - l_{20} \cdot x_2 - l_{30} \cdot x_3)/l_{00}$
21 // Store $X$ into memory
22 $X(3) \leftarrow x_3$
23 $X(2) \leftarrow x_2$
24 $X(1) \leftarrow x_1$
25 $X(0) \leftarrow x_0$
```

TABLE II: `div_ps()` and `sqrt_ps()` instruction latencies and throughputs on Haswell [7]

latency/throughtput	128-bit (*SSE*)	256-bit (*AVX*)
`···_add_ps()`	3/1	3/ 1
`···_mul_ps()`	5/0.5	5/ 0.5
`···_rcp_ps()`	5/1	7/ 2
`···_div_ps()`	11/7	19/14
`···_div_pd()`	16/8	28/16
`···_rsqrt_ps()`	5/1	7/ 2
`···_sqrt_ps()`	11/7	19/14
`···_sqrt_pd()`	16/8	28/16

instructions of data-independent loops (*Unroll&Jam*). In our case, it is applied on the outer loop.

This technique increases the register pressure with the order of unrolling k, the number of unrolled iterations. Unroll&jam of order k requires k times more local variables. Its efficiency is limited by the throughput of the unrolled loop instructions.

D. Fast square root reciprocal computation

`div_ps()` and `sqrt_ps()` are weakly pipelined and have a high throughput: 7 cycles with *SSE* (Table II). Both use the same execution unit and their throughput is shared between them. For example, a division can only be executed at least 7 cycles after a square root and vice versa.

The factorization of a $3{\times}3$ matrix requires 3 square roots and 3 divisions. Consequently, the factorization in *SSE* is at least $3 \times 7 + 3 \times 7 = 42$ cycles long for up to 4 matrices. The other 11 floating point operations have a lower throughput and can be partially executed in parallel to the square roots and divisions. Thus, their execution time is negligible compared to the square roots and divisions.

AVX square root and division have a throughput twice longer than *SSE*. The factorization is at $3 \times 14 + 3 \times 14 = 84$ cycles long for up to 8 matrices. In this case *AVX* is not faster than *SSE*.

These instructions can be replaced by faster and less accurate ones: `rcp_ps()` and `rsqrt_ps()`. They respectively compute an approximation of the reciprocal and an approximation of the square root reciprocal with a 12-bit accuracy. These instructions are highly pipelined and may improve the factorization speed. In our algorithm, only `rsqrt_ps()` is needed: $\sqrt{x} = x \cdot \sqrt{x}^{-1}$. A lot of cycles are saved using the fast instruction `rsqrt_ps()` instead of `sqrt_ps()` + `div_ps()`.

We do not need to compute any reciprocal as we get the reciprocal with the previous combination, and it is stored for later use. Every time the algorithm needs a division, a multiplication by the previously computed reciprocal is done.

It is less accurate than regular instructions, but it can be enough for some uses. If needed, it is possible to recover accuracy [12]. The method consists in the calculation of one Newton-Raphson iteration (algorithm 5). The Newton-Raphson algorithm converges quadratically which leads after one iteration to an accuracy of 0.46 *ulp* (Unit in last Place) in average on all normal finite positive `floats` (max: 4.7 *ulp*). Considering the latency only, it is actually slower than a regular square root. But it is much more pipelineable.

Algorithm 5: Accuracy recovering for $1/\sqrt{x}$

```
input  : $x$
output : $rsqrtx$   // approximate value of $1/\sqrt{x}$
1 $rsqrtx \leftarrow$ rsqrt_ps($x$)   // first approximation
2 $sqrtx \leftarrow x \cdot rsqrtx$
3 $\alpha \leftarrow sqrtx \cdot rsqrtx$
4 $rsqrtx \leftarrow -0.5\,(\alpha - 3) \cdot rsqrtx$   // corrected approximation
```

IV. Benchmarks

A. Benchmark protocol

In order to calculate the impact of the transforms, we used exhaustive benchmarks.

The algorithms were evaluated on 4 machines whose specifications are provided in Table III.

The tested functions are the following:

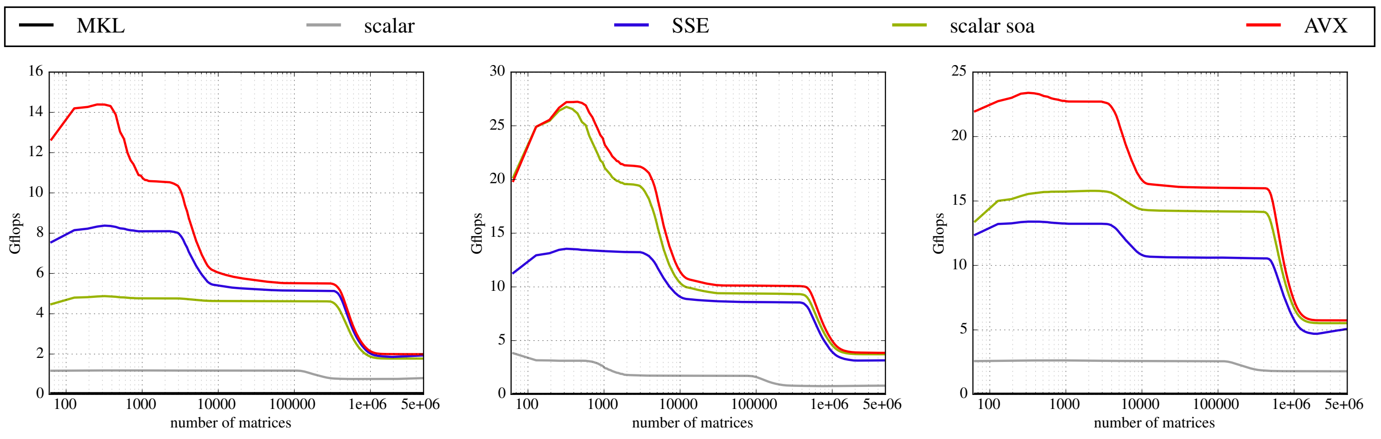
- **factorize:** Cholesky factorization: $A \rightarrow L \cdot L^T$
- **substitute:** Solve the 2 triangular systems: $L \cdot L^T \cdot X = R$
- **substitute1:** same as *substitute*, but with the same L for every Rs
- **solve:** Solve the unfactorized system (factorize + substitute): $A \cdot X = B$

The function *substitute1* has been tested as it is the only one to be available in the MKL in batch mode.

The time is measured with `_rdtsc()` which provides reliable time measures in cycles. Indeed, on current CPUs,

TABLE III: Benchmarked machines

CPU	fullname	cores/threads	cache (KiB) per core		per CPU	memory bandwidth (GB/s)		
			L1	*L2*	*L3*	1 core	1 CPU	total
i7	i7-4790	4/8	32	256	8192	7.8	7.8	
NHM	X5550	2× 4/8	32	256	8192	8.6	15.7	21.6
HSW	E5-2683 v3	2× 14/28	32	256	35840	10	40	75
KNC	7120P	61/244	32	512		5.3	300	



(a) *factorize* (b) *substitute* (c) *solve*

Fig. 3: Performance for 3×3 systems on HSW as a function of the number of matrices

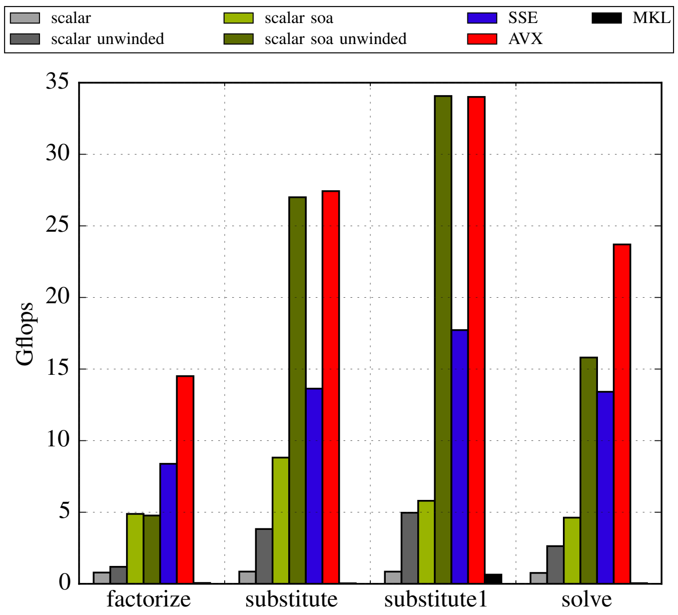


Fig. 4: Best performance for 3×3 systems on HSW

the timestamp counter is normalized around the nominal frequency of the processor and is independent from any frequency changes.

In order to have reliable measures, we run several times each function and take the minimum execution time measured. Then, we divide the time by the number of matrices to have a time per matrix.

The code has been compiled with intel icc v16.0.2 with the following options: `-std=c99 -O3 -vec -ansi-alias`

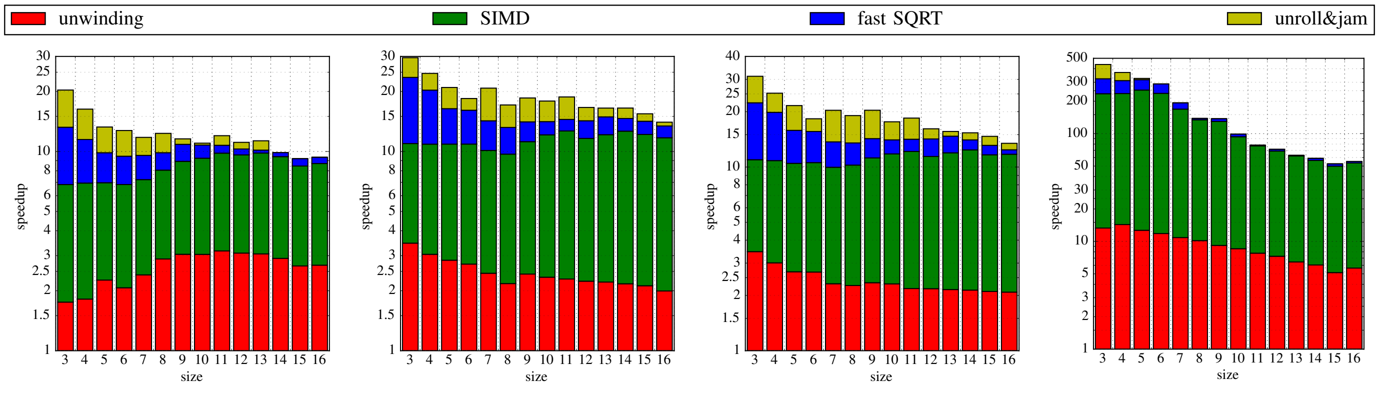
- Series labelled `scalar` are scalar written code. The *SoA* versions are vectorized by the compiler though.
- Series labelled `SSE` are *SSE* code executed on the machine, even if it is an *AVX* machine.
- Series labelled `AVX` are *AVX* code executed on the machine.
- "`unwinded`" tag stands for inner loops unwinded (ie: fully unrolled).
- "`fast`" tag stands for the use of fast square root reciprocal.
- "`×k`" tags stand for the order of unrolling of the outer loop (unroll&jam)

B. Results

We first present the monocore results and in a second part the multicore results with OPENMP.

We focus on the analysis of 3×3 matrix factorization as it is the slowest. We will show that our analysis remains correct for larger matrices.

1) Monocore: Figure 3 shows important results for the understanding of our function performance. It shows the performance of *factorize*, *substitute* and *solve* on HSW for 3×3 matrices. If we look at these charts, we can notice similar behaviors for the 3 functions: the performance drops of a factor 2-3 for every version. It happens when data do not fit anymore in caches: this is a cache overflow. On the *factorize*



(a) NHM (b) i7 (c) HSW (d) KNC

Fig. 5: Incremental speedups for *solve*

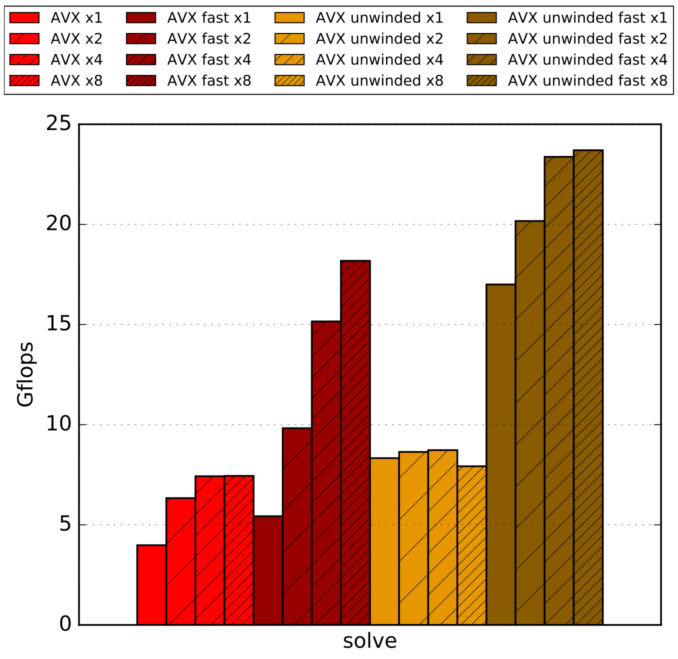


Fig. 6: Impact of loop unrolling for *solve* with 3×3 matrices on HSW

chart (Figure 3a), one can notice 3 intervals of batch size for 3×3 matrices on HSW:

- $[400, 1000]$: this is the *L1* cache overflow. As the *L1* cache is 32 KB, we can not store data for more than 546 systems.
- $[3000, 8000]$: this is the *L2* cache overflow. As the *L2* cache is 256 KB, we can not store data for more than 4369 systems.
- $[3{\cdot}10^5, 10^6]$: this is the *L3* cache overflow. As the *L3* cache is 35 MB, we can not store data for more than 611,669 systems. After that, the data has to be fetched from the main memory.

As we repeat several times the same function and take the minimum time, data are as much as possible within caches. If data fit within a cache, they may not be within it at the first execution, but they will at the next one. But if data size is larger than the cache, the cache will be constantly overflowed by new data. At the next execution, the needed data will not be within the cache as they have been overridden by the extra data of the previous execution. If data are only a bit larger than the cache, then a part can remain within the cache and be reused the next time.

Basically, one can interpret the performance plot like this: If all the matrices fit within the *L1* cache, the performance per matrix will be the performance on the plot before the *L1* cache overflow. The performance at the right end is actually the performance when none of the matrices are in any caches, ie: they are in main memory only. The performance drops after the cache overflow because lower level caches are faster.

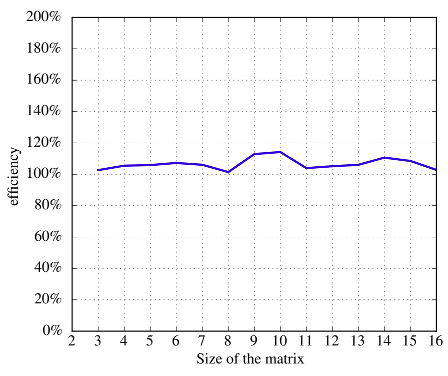
After the *L3* cache overflow, the best versions have almost the same performances: they are limited by the memory bandwidth. In this case, the bandwidth of the *factorize* function after the last cache overflow is about 10 GB/s, which the bandwidth of our machine external memory .

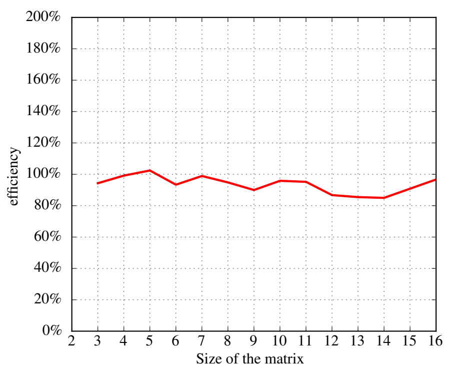
On every plot of Figure 3, for the fastest version, the performance starts by increasing on the left. This is mainly due to the amortization of the overheads mainly due to *SIMD*.

The MKL is very slow. The reason is that it does a lot of verification on input data, has many functions calls and has huge overheads. These overheads are required for speeding up the execution, but are effective only for large matrices. For large matrices, the MKL is, of course, very fast and it would have impossible to be faster than the MKL for large matrices.

Still on Figure 3, the scalar *AoS* versions are slow: they are not vectorized, unlike the other versions. The compiler is unable to vectorize these because of the memory layout even with `-vec` compiler option. Although, it is possible to write *SIMD* code for *AoS*, but requires scatter/gather instructions (or a way to emule it).

When we look at Figure 4, we can see that the scalar *SoA*


(a) NHM


(b) HSW

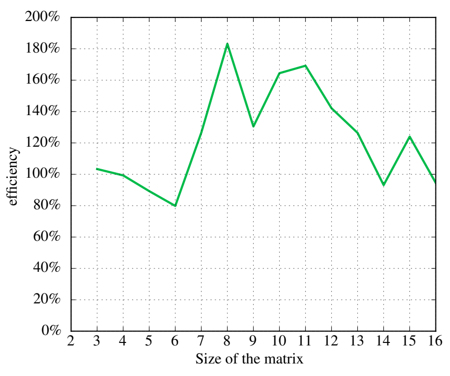
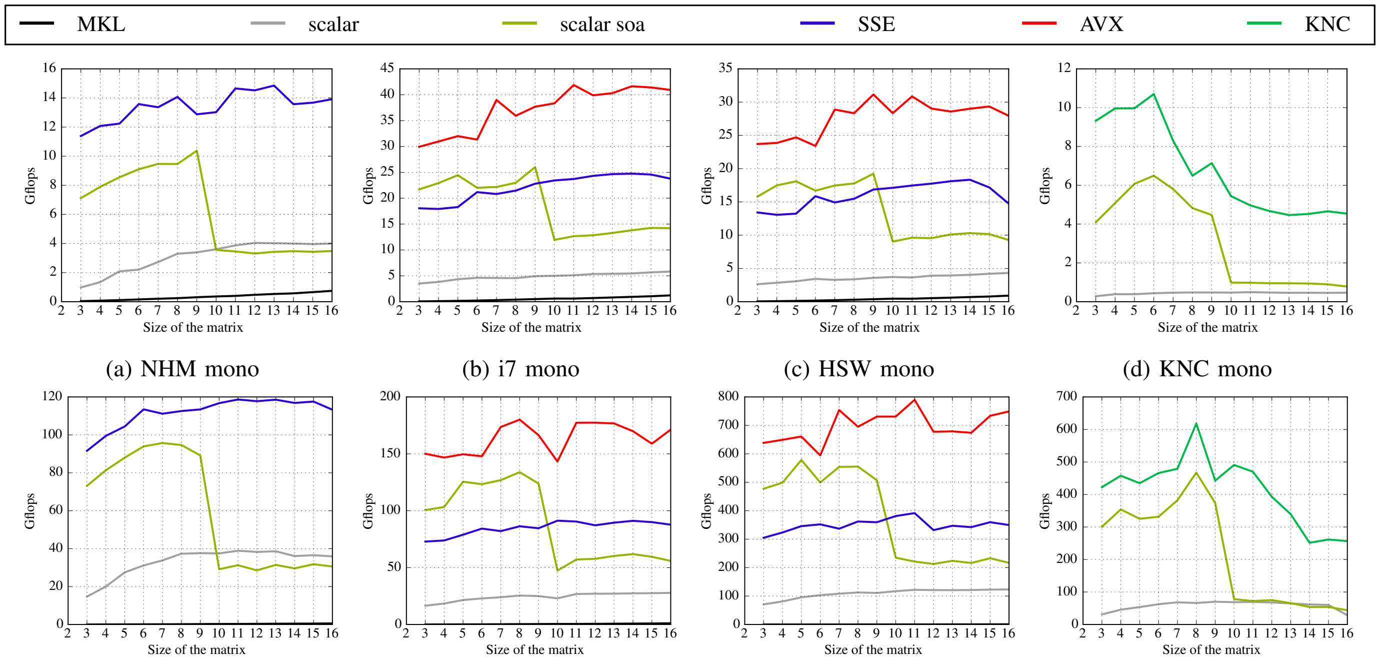

(c) KNC

Fig. 7: Efficiency of *solve* multithreading



(a) NHM mono (b) i7 mono (c) HSW mono (d) KNC mono

(e) NHM OpenMP (f) i7 OpenMP (g) HSW OpenMP (h) KNC OpenMP

Fig. 8: Best performance of *solve* for multiple sizes

unwinded version performs very well on *substitute* and *substitute1*, but is slower on other functions. The function *substitute1* provides the higher Gflops: the number of load/store is the lowest as L is kept in registers. The plots for other sizes and other machines have similar shapes and are not shown here.

Figure 5 gives the speedup of each transformation in the following order: unwinding, *SoA* + *SIMD*, fast square root, unroll&jam. The speedup of a transformation is dependent of the transformations already applied: the order is important.

The speedup given by the KNC plot (Figure 5d) is actually not representative as KNC is not adapted for scalar code. However, this plot allows to see the impact of each transformation on the performance.

If we look at the speedups on HSW (Figure 5c), we can see that unwinding the inner loops improves the performance well: from $\times 2$ to $\times 3.5$. The impact of unwinding decreases when the size of the matrix increase: the register pressure is higher. *SIMD* gives a sub-linear speedup: from $\times 3.2$ to $\times 5.5$. In fact, *SIMD* instructions can not be fully efficient on this function without fast square root (see subsection III-D). With further analysis, we can see that the speedup of *SIMD* + fast square root is almost constant around $\times 6$. The impact of the fast square root decreases as their number become negligible compared to the other floating point operations. *SIMD* has more place to be efficient. For small matrices, unroll&jam allows to get the last part of the expected *SIMD* speedup. *SIMD* + fast square root + unroll&jam: from $\times 6.5$ to $\times 9$. Unroll&jam loses its efficiency for larger matrices: the register pressure is higher.

Figure 6 shows the performance of *solve* for different *AVX* versions. The performance of the "non-fast" versions are limited by the square roots and divisions around 9 Gflops (see subsection III-D). In this case, both unrolling (unwinding and unroll&jam) are not very efficient and can not improve performance further this limitation. For "fast" versions, both unrolling are efficient. Unroll&jam achieves a $\times 3$ speedup on regular code and $\times 1.5$ speedup on unwinded code. We can see that unroll&jam is less efficient when the code is already unwinded but keeps improving the performance. Register pressure is higher when unrolling (unwinding or unroll&jam).

2) OpenMP: Figure 7 shows how our implementation scales with higher number of cores. The scaling of the *SIMD* version is strong and for some sizes, the code get a super-linear speedup.

The MKL version however does not scale at all: it does not support batch function for the Cholesky factorization and it does not enable the multithreading as the matrices are too small according to its heuristics. Again, the MKL is not adapted for tiny matrices, but scales very well for large matrices.

On KNC, the scaling is strong and for some sizes of matrices, it has a super-linear speedup (see Figure 7c). The scaling is actually not representative as it is compared to mono-thread execution. Indeed, latencies on KNC are high and a single thread can not use efficiently the whole core.

3) Extension to matrices up to 16×16*:* Figure 8 shows that our transformations studied for 3×3 matrices are also efficient with larger matrices. Speedups for benched machines are summarized into Table IV. Again, the speedup for KNC is not representative as it is compared to scalar code.

TABLE IV: Speedups of *solve* for matrices up to 16×16

machine	speedup monocore	speedup multicore
i7	$\times14 - \times30$	$\times14 - \times29$
NHM	$\times8 - \times21$	$\times9 - \times19$
HSW	$\times13 - \times31$	$\times15 - \times33$
KNC	$\times40 - \times440$	$\times50 - \times170$

Conclusion

In this paper, we have presented a fast implementation for tiny matrices ($\leqslant 16\times16$) of the well-known Cholesky algorithm. The State-of-the-Art codes and libraries lack of support for tiny matrices: existing libraries deals with large matrices (usually thousands of rows), and are usually slow for small matrices. For a lot of people, a "small matrix" is about few hundreds rows. But in Computer Vision/High Energy Physics, people need tiny matrices and scalar naive code was a rather good option. We have shown that the scalar naive code is much slower compared to the very specialized code we wrote.

Our approach is to compute the factorization by batch which enables efficient parallelization (both *SIMD* + OpenMP). We achieve a high overall speedup: more than an order of magnitude compared to scalar naive version in monothread. The speedup for 3×3 matrices on 2×14 cores Haswell Xeon is between $\times13$ and $\times31$. Our multi-threaded implementation has almost the same speedup compared to the multithread naive code. Our Haswell get a speedup between $\times15$ and $\times33$

The speedup decreases as the reference version becomes faster, but for all sizes of matrices from 3×3 to 16×16, the code performance is close to the sustainable peak performance of the processor.

In the future, we will add support for other *SIMD* architectures like ARM Neon or IBM Altivec. The double precision and mixed precision will also be part of futur implementations.

References

[1] J. Abel, K. Balasubramanian, M. Bargeron, T. Craver, and M. Phlipot. Applications tuning for streaming SIMD extensions. *Intel Technology Journal*, 1999.

[2] R. Allen and K. Kennedy, editors. *Optimizing compilers for modern architectures: a dependence-based approach*, chapter 8,9,11. Morgan Kaufmann, 2002.

[3] ATLAS. Automatically tuned linear algebra software http://math-atlas.sourceforge.net.

[4] D. Beymer, P. McLauchlan, B. Coifman, and J. Malik. A real-time computer vision system for measuring traffic parameters. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pages 495–501. IEEE, 1997.

[5] T. Dong, A. Haidar, P. Luszczek, J. A. Harris, S. Tomov, and J. Dongarra. LU factorization of small matrices: accelerating batched DGETRF on the GPU. In *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on*, pages 157–160. IEEE, 2014.

[6] T. Dong, A. Haidar, S. Tomov, and J. Dongarra. A fast batched cholesky factorization on a GPU. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 432–440. IEEE, 2014.

[7] A. Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, 2016. accessed version: 2016-01-09.

[8] R. Frühwirth. Application of Kalman filtering to track and vertex fitting. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 262(2):444–450, 1987.

[9] N. J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002.

[10] N. J. Higham. Cholesky factorization. *Wiley Interdisciplinary Reviews: Computational Statistics*, 1(2):251–254, 2009.

[11] J. Iliffe. The use of the genie system in numerical calculation. *Annual Review in Automatic Programming*, 2:1–28, 1961.

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-026. April 2012.

[13] L. Lacassagne, D. Etiemble, A. Hassan-Zahraee, A. Dominguez, and P. Vezolle. High level transforms for SIMD and low-level computer vision algorithms. In *ACM Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*, pages 49–56, 2014.

[14] A. Romero, L. Lacassagne, and M. Gouiffès. Real-time covariance tracking algorithm for embedded systems. In *IEEE International Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2013.

[15] J. Shin, M. W. Hall, J. Chame, C. Chen, and P. D. Hovland. Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology. In *Software Automatic Tuning*, pages 353–370. Springer, 2011.

[16] SPIRAL. Spiral: Sotfware/hardware generation for dsp algorithms http://www.spiral.net.

[17] X. Tian, H. Saito, S. V. Preis, E. N. Garcia, S. S. Kozhukhov, M. Masten, A. G. Cherkasov, and N. Panchenko. Effective SIMD vectorization for Intel Xeon Phi coprocessors. *Scientific Programming*, 501:269764, 2015.