Cholesky Factorization on SIMD multi-core architectures

Florian Lemaitre, Benjamin Couturier, Lionel Lacassagne

Cholesky Factorization on *SIMD* multi-core architectures

Florian Lemaitre [1,2] Benjamin Couturier [1] Lionel Lacassagne [2]
[1] CERN on behalf of the LHCb Collaboration, Geneva, Switzerland
[2] Sorbonne Universites, UPMC Univ Paris 06, CNRS UMR 7606, LIP6, Paris, France
florian.lemaitre@cern.ch – ben.couturier@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 speed up 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 multi-core and many-core *SIMD* architectures (*SSE*, *AVX*2, *KNC*, *AVX512*, Neon, Altivec). We focus on the fact that, on some architectures, compilers are unable to vectorize and on other architectures, vectorizing compilers are not efficient. Thus hand-made *SIMD*ization is mandatory. We achieve with these transformations combined with *SIMD* a speedup from $\times 14$ to $\times 28$ for the whole resolution in single precision compared to the naive code on a *AVX*2 machine and a speedup from $\times 6$ to $\times 14$ on double precision, both with a strong scalability.**

I. Introduction

Linear algebra is everywhere, especially in scientific computation. There are a lot of fast linear algebra libraries like MKL [1], Magma [2] or Eigen [3]. However, these libraries are optimized for big matrices. Experience shows that these libraries are not adapted for tiny matrices and perform slow on them.

Many computer vision applications require real-time processing, especially for autonomous robots or associated to statistical figures, linear and curves fitting, ellipse fitting or even covariance matching [4]. This is also the case in High Energy Physics (HEP) where computation should be done on-the-fly. In these domains, it is usual to manipulate tiny matrices. There is, therefore, a need for a linear algebra which is different from classical High Performance Computing. Matrices up to a few dozen of rows are usual, for example through Kalman Filter: [5] uses a 5 dimensions Kalman Filter and [6] uses 4 dimensions. More and more people take an interest in smaller and smaller matrices [7], [8], [9].

The goal of this paper is to present an optimized implementation of a linear system solver for tiny matrices using the Cholesky factorization on *SIMD* multi-core architectures, for which it exists no efficient implementation unlike on GPUs [10]. People tend to rely on the compiler to vectorize the scalar code, but the result is not efficient and can be improved by manually writing *SIMD* code.

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 inside one single factorization. Our approach is similar to Spiral [11] or ATLAS [12]: we compare many different implementations of the same algorithm to keep the best one for each architecture.

We expose first the Cholesky algorithm in section II. Then we explain the transformations we made to improve the performance for tiny matrices in section III. We discuss about the precision and the accuracy of the square root and how use these considerations to improve our implementation in section IV. And finally, we present the result of the benchmarks in section V.

II. Cholesky Algorithm

The whole resolution is composed of 3 steps: the Cholesky factorization (also known as 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 terms of floating-point operations of $n^3/3$ that is half of the LU one ($2n^3/3$), and is numerically more stable [13], [14]. 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. It requires n square roots and $(n^2 + 3n)/2$ divisions for $n \times n$ matrices which are slow operations especially on double precision.

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

Algorithm 1: Cholesky Factorization

input : A // $n \times n$ symmetric positive-definite matrix
output : L // $n \times n$ lower triangular matrix

for $j = 0 : n-1$ **do**
  $s \leftarrow A(j,j)$
  for $k = 0 : j-1$ **do**
    $s \leftarrow s - L(j,k)^2$
  $L_{j,j} \leftarrow \sqrt{s}$
  for $i = j+1 : n-1$ **do**
    $s \leftarrow A(i,j)$
    for $k = 0 : j-1$ **do**
      $s \leftarrow s - L(i,k) \cdot L(j,k)$
    $L(i,j) \leftarrow s / L(j,j)$

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**
    $s \leftarrow s - L(j,i) \cdot X(j)$
  $X(i) \leftarrow s / L(i,i)$

TABLE I: Number of floating-point operations

(a) Classic: with array access

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) Optimized: with scalarization and reuse

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	$\frac{1}{6}\left(2n^3 + 15n^2 + 7n\right)$	$\frac{1}{2}\left(n^2 + 6n\right)$	$\sim 2n/3$

C. Batch

With small matrices, parallelization is not efficient as there is no long dimension. For instance, a 3-iteration loop cannot be efficiently vectorized.

The idea is to add one extra and long dimension to compute the Cholesky factorization of a large set of matrices instead of one. We can now parallelize along this dimension with both vectorization and multithreading. 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 [15].

III. Transformations

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

- High Level Transforms: memory layout [16] and fast square root (the latter is detailed in section IV),
- loop transforms (loop unwinding [17], loop unrolling and unroll&jam),
- Architectural transforms: *SIMD*ization.

With all these transformations, the number of possible versions is high. More specially, loop unwinding generates different versions for each matrix size. To facilitate this, the code is automatically generated for all transformations and all sizes from 3×3 up to 16×16 with the template engine Jinja2 [18] in Python. It generates *C99* code with the `restrict` keyword which helps the compiler to vectorize. This could be replaced by a *C++* template metaprogram like in [19].

The use of jinja2 instead of more common meta-programmation methods allows us to have full access and control over the generated code. It is really important for some people like in embedded systems to have access to the source code before the compilation. They can understand more easily some bugs which are hard to track on black box systems.

A. Memory Layout Transform

The memory layout transform is the first transform to address as the other ones rely on it. The most important aspect of the memory layout is the battle between *AoS* (Array of Structures) and *SoA* (Structure of arrays) [20] (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 member, 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 (Array of *SoA*, also known as Hybrid *SoA*) tries to combine the advantages of *AoS* and *SoA* for *SIMD*. 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.

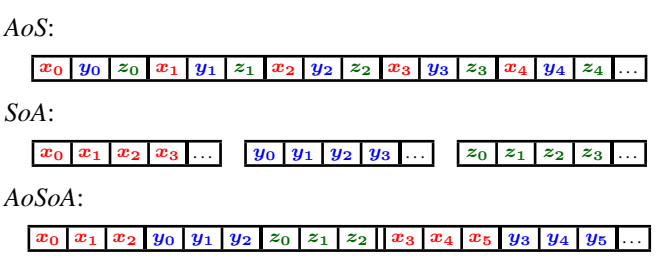


Fig. 1: Memory Layouts

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`, `_mm_malloc` or `aligned_alloc` (in *C11*). 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 $N\,M$ 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 [21] 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 store arrays of variable length rows like triangular matrices or padded/shifted arrays and remains completely transparent to the user at they will always access $A(i,j)$ with `A[i][j]` whatever is used internally, as long as $A(i,j)$ is mathematically correct. It is 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 from the previous address.

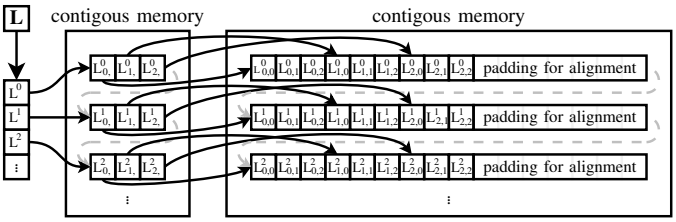


Fig. 2: Iliffe vector example: array of 3×3 matrices aligned with padding

B. Loop unwinding

Loop unwinding is the special case of loop unrolling where the loop is entirely unrolled. It is possible to do it here as the matrices are tiny (see algorithms 3 and 4). 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,

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. When the factorization and the substitution are merged together and scalarized, even more memory accesses can be removed: storing L (lines 18–21 of algorithm 3) and loading L again (lines 2–5 of algorithm 4) are unnecessary as L registers are still available. This leads to algorithm 5 and reduces the amount of memory accesses (Table Ib).

The register pressure is higher and the compiler may generate spill code to temporarily store variables into memory.

To facilitate the unwinding for multiple sizes, the code is automatically generated for any given sizes by Jinja2.

Algorithm 3: Factorization unwinded for 4×4 matrices

input : A // 4×4 symmetric positive-definite matrix
output : L // 4×4 lower triangular matrix

// Load A into registers
$a_{00} \leftarrow A(0,0)$
$a_{10} \leftarrow A(1,0) \quad a_{11} \leftarrow A(1,1)$
$a_{20} \leftarrow A(2,0) \quad a_{21} \leftarrow A(2,1) \quad a_{22} \leftarrow A(2,2)$
$a_{30} \leftarrow A(3,0) \quad a_{31} \leftarrow A(3,1) \quad a_{32} \leftarrow A(3,2) \quad a_{33} \leftarrow A(3,3)$
// Factorize A
$l_{00} \leftarrow \sqrt{a_{00}}$
$l_{10} \leftarrow a_{10}/l_{00}$
$l_{20} \leftarrow a_{20}/l_{00}$
$l_{30} \leftarrow a_{30}/l_{00}$
$l_{11} \leftarrow \sqrt{a_{11} - {l_{10}}^2}$
$l_{21} \leftarrow (a_{21} - l_{20} \cdot l_{10})/l_{11}$
$l_{31} \leftarrow (a_{31} - l_{30} \cdot l_{10})/l_{11}$
$l_{22} \leftarrow \sqrt{a_{22} - {l_{20}}^2 - {l_{21}}^2}$
$l_{32} \leftarrow (a_{32} - l_{30} \cdot l_{20} - l_{31} \cdot l_{21})/l_{22}$
$l_{33} \leftarrow \sqrt{a_{33} - {l_{30}}^2 - {l_{31}}^2 - {l_{32}}^2}$
// Store L into memory
$L(0,0) \leftarrow l_{00}$
$L(1,0) \leftarrow l_{10} \quad L(1,1) \leftarrow l_{11}$
$L(2,0) \leftarrow l_{20} \quad L(2,1) \leftarrow l_{21} \quad L(2,2) \leftarrow l_{22}$
$L(3,0) \leftarrow l_{30} \quad L(3,1) \leftarrow l_{31} \quad L(3,2) \leftarrow l_{32} \quad L(3,3) \leftarrow l_{33}$

Algorithm 4: Substitution unwinded for 4×4 matrices

input : L // 4×4 lower triangular matrix
input : R // vector of size 4
output : X // vector of size 4, solution of $L \cdot L^T \cdot X = R$

// Load L into registers
$l_{00} \leftarrow L(0,0)$
$l_{10} \leftarrow L(1,0) \quad l_{11} \leftarrow L(1,1)$
$l_{20} \leftarrow L(2,0) \quad l_{21} \leftarrow L(2,1) \quad l_{22} \leftarrow L(2,2)$
$l_{30} \leftarrow L(3,0) \quad l_{31} \leftarrow L(3,1) \quad l_{32} \leftarrow L(3,2) \quad l_{33} \leftarrow L(3,3)$
// Load R into registers
$r_0 \leftarrow R(0) \quad r_1 \leftarrow R(1) \quad r_2 \leftarrow R(2) \quad r_3 \leftarrow R(3)$
// Forward substitution
$y_0 \leftarrow r_0/l_{00}$
$y_1 \leftarrow (r_1 - l_{10} \cdot y_0)/l_{11}$
$y_2 \leftarrow (r_2 - l_{20} \cdot y_0 - l_{21} \cdot y_1)/l_{22}$
$y_3 \leftarrow (r_3 - l_{30} \cdot y_0 - l_{31} \cdot y_1 - l_{32} \cdot y_1)/l_{33}$
// Backward substitution
$x_3 \leftarrow y_3/l_{33}$
$x_2 \leftarrow (y_2 - l_{32} \cdot x_3)/l_{22}$
$x_1 \leftarrow (y_1 - l_{21} \cdot x_2 - l_{31} \cdot x_3)/l_{11}$
$x_0 \leftarrow (y_0 - l_{10} \cdot x_1 - l_{20} \cdot x_2 - l_{30} \cdot x_3)/l_{00}$
// Store X into memory
$X(3) \leftarrow x_3 \quad X(2) \leftarrow x_2 \quad X(1) \leftarrow x_1 \quad X(0) \leftarrow x_0$

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 (also known as throughput) is smaller than the latency. The latency of pipelined instructions

Algorithm 5: Cholesky factorization + substitution unwinded and scalarized for 4×4 matrices

input : A // 4×4 symmetric positive-definite matrix
input : R // vector of size 4
output : X // vector of size 4, solution of $L \cdot L^T \cdot X = R$

// Load A into registers
$a_{00} \leftarrow A(0,0)$
$a_{10} \leftarrow A(1,0) \quad a_{11} \leftarrow A(1,1)$
$a_{20} \leftarrow A(2,0) \quad a_{21} \leftarrow A(2,1) \quad a_{22} \leftarrow A(2,2)$
$a_{30} \leftarrow A(3,0) \quad a_{31} \leftarrow A(3,1) \quad a_{32} \leftarrow A(3,2) \quad a_{33} \leftarrow A(3,3)$
// Load R into registers
$r_0 \leftarrow R(0) \quad r_1 \leftarrow R(1) \quad r_2 \leftarrow R(2) \quad r_3 \leftarrow R(3)$
// Factorize A
$l_{00} \leftarrow \sqrt{a_{00}}$
$l_{10} \leftarrow a_{10}/l_{00}$
$l_{20} \leftarrow a_{20}/l_{00}$
$l_{30} \leftarrow a_{30}/l_{00}$
$l_{11} \leftarrow \sqrt{a_{11} - {l_{10}}^2}$
$l_{21} \leftarrow (a_{21} - l_{20} \cdot l_{10})/l_{11}$
$l_{31} \leftarrow (a_{31} - l_{30} \cdot l_{10})/l_{11}$
$l_{22} \leftarrow \sqrt{a_{22} - {l_{20}}^2 - {l_{21}}^2}$
$l_{32} \leftarrow (a_{32} - l_{30} \cdot l_{20} - l_{31} \cdot l_{21})/l_{22}$
$l_{33} \leftarrow \sqrt{a_{33} - {l_{30}}^2 - {l_{31}}^2 - {l_{32}}^2}$
// Forward substitution
$y_0 \leftarrow r_0/l_{00}$
$y_1 \leftarrow (r_1 - l_{10} \cdot y_0)/l_{11}$
$y_2 \leftarrow (r_2 - l_{20} \cdot y_0 - l_{21} \cdot y_1)/l_{22}$
$y_3 \leftarrow (r_3 - l_{30} \cdot y_0 - l_{31} \cdot y_1 - l_{32} \cdot y_1)/l_{33}$
// Backward substitution
$x_3 \leftarrow y_3/l_{33}$
$x_2 \leftarrow (y_2 - l_{32} \cdot x_3)/l_{22}$
$x_1 \leftarrow (y_1 - l_{21} \cdot x_2 - l_{31} \cdot x_3)/l_{11}$
$x_0 \leftarrow (y_0 - l_{10} \cdot x_1 - l_{20} \cdot x_2 - l_{30} \cdot x_3)/l_{00}$
// Store X into memory
$X(3) \leftarrow x_3 \quad X(2) \leftarrow x_2 \quad X(1) \leftarrow x_1 \quad X(0) \leftarrow x_0$

can be hidden by executing another instruction in the pipeline without any data-dependence with the previous one. The *ipc* (instructions per cycle) is then limited by the throughput[1] of the instruction and not by its latency. 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 the 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 instructions of data-independent loops (Unroll&Jam). Here, Unroll&Jam of factor 2, 4 and 8 is applied to the outer loop over the array of matrices.

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.

[1] 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.

TABLE II: *SIMD* instruction latencies and throughputs for single and double precision instructions on Haswell [22]

latency/throughput	128-bit (*SSE*)	256-bit (*AVX*)
···_cvtps_pd()	2/1	5/ 1
···_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**
···_rsqrt_ps()	5/1	7/ 2
···_sqrt_ps()	11/**7**	19/**14**
···_cvtpd_ps()	4/1	5/ 1
···_add_pd()	3/1	3/ 1
···_mul_pd()	5/0.5	5/ 0.5
···_div_pd()	16/**8**	28/**16**
···_sqrt_pd()	16/**8**	28/**16**

Unroll&jam is also generated by our Jinja2 templates.

IV. Precision and Accuracy

The Cholesky Algorithm requires n square roots and $(n^2+3n)/2$ divisions for a $n\times n$ matrix. But these arithmetic operations are slow, especially for double precision (Table II) and usually not fully pipelined. For example, divisions and square root require 7 cycles for single precision in *SSE*. The cycle penalty for these operations reaches 16 cycles for double precision in *AVX*. During these cycles, no other instructions can be executed in the same pipeline port, even with hyperthreading. Thus, square roots and divisions limit the overall Cholesky throughput.

As explained by Soderquist [23], it is possible in hardware to compute them faster with less accuracy. That is why reciprocal functions are available: they are faster but have a lower accuracy: usually 12 bits for a 23-bit mantissa in single precision.

The accuracy is measured in *ulp* (Unit in Last Place). Given a floating-point number x, $ulp(x)$ is the distance between x and the floating-point number that come just after x. For all normal floating-point numbers $ulp(x) = ulp(x+\epsilon)$ iif x and $x+\epsilon$ have the same exponent ($\lfloor \log_2(x) \rfloor = \lfloor \log_2(x+\epsilon) \rfloor$, power of 2 are the corner cases). In this case, one can omit which number *ulp* refer to.

A. Memorization of the reciprocal value

In the algorithm, a square root is needed to compute $L(i,i)$. But $L(i,i)$ is used in the algorithm only with divisions. The algorithm needs $\left(n^2+3n\right)/2$ of these divisions per $n\times n$ matrix.

Instead of computing $x/L(i,i)$, one can compute $x \cdot L(i,i)^{-1}$. It becomes obvious that one can store $L(i,i)^{-1}$ and save several divisions. After this transformation, the algorithm needs only n divisions.

This transformation might affect the accuracy of the result. Indeed, x/y is rounded once (correct rounding as specified by IEEE 754). But $x \cdot \left(y^{-1}\right)$ requires two successive roundings: one to compute the reciprocal $y^{-1} = 1/y$ and the other one to compute the product $x \cdot \left(y^{-1}\right)$. Thus, $x \cdot \left(y^{-1}\right)$ has an error < 1 *ulp* instead of < 0.5 *ulp* when computed directly.

B. Fast square root reciprocal estimation

The algorithm performs a division by a square root and therefore needs to compute $f(x) = 1/\sqrt{x}$. There are some ways to compute an estimation of this function depending on the precision.

TABLE III: Square root reciprocal estimate instructions

ISA	intrinsic name	error	machines
Neon	vrsqrteq_f32	$< 2^{-12}$	A53, A57
Altivec	vec_rsqrte	$< 2^{-12}$	P6 $\rightarrow$ P8
SSE	_mm_rsqrt_ps	$< 2^{-12}$	*NHM*
AVX	_mm256_rsqrt_ps	$< 2^{-12}$	*SDB* $\rightarrow$ *SKL*
KNCNI	_mm512_rsqrt23_ps	< 0.5 *ulp*	*KNC*
AVX512F	_mm512_rsqrt14_ps	$< 2^{-14}$	*SKL* Xeon
AVX512ER	_mm512_rsqrt28_ps	< 0.5 *ulp*	*KNL*

1) Single Precision: Most of current CPUs have a specific instruction to compute an estimation of the square root reciprocal in single precision. In fact, some *ISA* (Instruction Set Architecture) like Neon and Altivec *VMX* do not have *SIMD* instruction for the square root and the division, but do have an instruction for a square root reciprocal estimation. On x86, ARM and Power, this instruction is as fast as the multiplication (Table II) and gives an estimation with 12-bit accuracy (Table III). Unlike regular square root and division, this instruction is fully pipelined ($throughput = 1$) and thus avoids pipeline stall.

Algorithm 6: Double Precision RSQRT estimate (through single precision) 12-bit accurate

input : $x_{0,F64}$, $x_{1,F64}$
output: $\hat{r}_{0,F64}$, $\hat{r}_{1,F64}$ // estimation of $1/\sqrt{x}$
1 $\text{low}(x_{F32}) \leftarrow \text{convert\_f64\_to\_f32}(x_{0,F64})$
2 $\text{high}(x_{F32}) \leftarrow \text{convert\_f64\_to\_f32}(x_{1,F64})$
3 $\hat{r}_{F32} \leftarrow \text{rsqrte}(x_{F32})$ // single precision 12-bit estimate
4 $\hat{r}_{0,F64} \leftarrow \text{convert\_f32\_to\_f64}(\text{low}(\hat{r}_{F32}))$
5 $\hat{r}_{1,F64} \leftarrow \text{convert\_f32\_to\_f64}(\text{high}(\hat{r}_{F32}))$

2) Double Precision: On most CPUs, there is no such instruction for double precision (only *AVX512F* has such). Therefore, we need another way to get an estimate. A possibility is to convert two *SIMD* double precision registers into a single single precision register and execute the single precision instruction to get a 12-bit accurate estimation and convert the

Algorithm 7: Double Precision RSQRT estimate (bit trick)

input : x_{F64}
output: $\hat{r}_{F64}$ // estimation of $1/\sqrt{x}$

$x_{I64} \leftarrow \text{cast\_f64\_to\_i64}(x_{F64})$
$\hat{r}_{I64} \leftarrow \texttt{0x5fe6eb50c7b537a9} - (x_{I64} >> 1)$
$\hat{r}_{F64} \leftarrow \text{cast\_i64\_to\_f64}(\hat{r}_{I64})$

register back into two double precision registers (algorithm 6). This technique has a constraint: it can be used only if the input is within the range of single precision floating-point $\left[2^{-126}, 2^{127}\right]$ ($\left[\sim 10^{-38}, \sim 10^{38}\right]$). Cholesky algorithm needs to compute the square root of a difference, so if this difference is very close to 0, catastrophic cancellation may occur, and the value may not be in the range of single precision float. This issue is not handled by this approach.

3) Bit Trick for Double Precision: The square root reciprocal can be estimated directly in double precision taking benefits from the IEEE 754 floating-point format (algorithm 7). This is mathematically explained by Lomont in [24]. It was initially attributed to John Carmack in the Quake III Arena source code. A quick explanation could be like this: the right bit shift allows to divide by 2 the exponent (effect of the square root) and the subtraction allows to take the opposite of the exponent (effect of the reciprocal). The rest of the magic constant `0x5fe6eb50c7b537a9` is set up to minimize the error of the result as explained by Lomont. This technique is really fast (especially when integer and floating-point operations can be executed in parallel by the CPU) but inaccurate ($\epsilon \sim 0.0342128 \sim \frac{1}{29}$).

C. Accuracy recovering

Depending on the application, the previous techniques might not be accurate enough (especially the bit trick in double precision subsubsection IV-B3). It is possible to recover the accuracy with the Newton-Rahpson method or the Householder's method (a higher order generalization). It is worth noting that if one does not require full accuracy, they can reduce the number of iterations done in order to be faster.

Algorithm 8: Newton Raphson for $1/\sqrt{x}$

input : x
input : $\hat{r}$ // estimation of $1/\sqrt{x}$
output: r // corrected estimation

$\alpha \leftarrow \hat{r} \cdot \hat{r} \cdot x$
$r \leftarrow 0.5 \cdot \hat{r} \cdot (3 - \alpha)$ // corrected approximation

1) Newton-Raphson: The Newton-Raphson method is an iterative algorithm to find roots of a function $f(x)$. Given an estimation x_n of a root of f, one can find a more accurate estimation x_{n+1} with the following formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1}$$

This method can be used to refine an estimation of a square root reciprocal. To compute the square root reciprocal of a, one can find the root of $f(x) = \frac{1}{x^2} - a$. Applying (1) to this function gives the following equation:

$$x_{n+1} = \frac{1}{2}\, x_n \left(3 - {x_n}^2\, a\right) \tag{2}$$

With the equation (2), the iteration needs 4 multiplications (algorithm 8). But one can see that the multiplication by $1/2$ can be moved inside the brackets and the product $1/2 \cdot a$ computed once before any iteration. After this, it requires a multiplication at initialization plus 3 multiplications and one subtraction per iteration. The subtraction can be fused into a Fused Multiply Add (FMA) if supported by the architecture.

The Newton-Raphson method has a quadratic convergence. This means that the number of correct bits doubles at each iteration (ϵ becomes ϵ^2). With the single precision estimate, one iteration is needed and allows to recover almost full accuracy with a mean error < 0.5 *ulp* and a max error < 4.7 *ulp* ($\sim$2.5 bits). The maximum and mean relative error are computed by computing relative error with algorithm 9 exhaustively on all normal single precision floats. For double precision, results are reported in Table IV.

The Neon *ISA* supports an instruction to help Newton-Raphson method for $f(x) = \frac{1}{x^2} - a$ (algorithm 10). The instruction `vrsqrtsq_f32` is as fast as a multiplication and

Algorithm 9: Relative error of $\texttt{rsqrt}(x)$ in *ulp*

input : x_{F32}
output: ϵ_{32}

// compute 12-bit estimate + 1 Newton-Raphson iteration (F32)
$\hat{r}_{F32} \leftarrow \texttt{rsqrt}(x)$
$x_{F64} \leftarrow \text{convert\_f32\_to\_f64}(x)$
$\hat{r}_{F64} \leftarrow 1/\sqrt{x_{F64}}$ // F64 computation
$\hat{r}_{I64} \leftarrow \text{cast\_f64\_to\_i64}(\hat{r}_{F64})$
$\hat{r}_{F32 \to F64} \leftarrow \text{convert\_f32\_to\_f64}(\hat{r}_{F32})$
$\hat{r}_{F32 \to I64} \leftarrow \text{castf64\_to\_i64}(\hat{r}_{F32 \to F64})$
$\epsilon_{64} \leftarrow \left|\hat{r}_{I64} - \hat{r}_{F32 \to I64}\right|$ // F64 *ulp*
$\epsilon_{32} \leftarrow \epsilon_{64}/2^{53-24}$ // F32 *ulp*

Algorithm 10: Newton Raphson for $1/\sqrt{x}$ with Neon

input : x
input : $\hat{r}$ // estimation of $1/\sqrt{x}$
output: r // corrected estimation

$\alpha \leftarrow \texttt{vrsqrtsq\_f32}(\hat{r} \cdot x,\, \hat{r})$
$r \leftarrow \hat{r} \cdot \alpha$ // corrected approximation

TABLE IV: Newton-Raphson error recovery

source prec	target prec	#iter	#FMA	#mul	#add
$1/29$ (bit trick)	2^{-24} (F32)	3	3	7	-
2^{-12}	2^{-24} (F32)	1	1	3	-
2^{-14}	2^{-24} (F32)	1	1	3	-
$1/29$ (bit trick)	2^{-53} (F64)	4	4	9	-
2^{-12}	2^{-53} (F64)	3	3	7	-
2^{-14}	2^{-53} (F64)	2	2	5	-
2^{-23}	2^{-53} (F64)	2	2	5	-
2^{-28}	2^{-53} (F64)	1	1	3	-
initialization step			-	1	-
single iteration			1	2	-

TABLE V: Householder's method orders for full accuracy recovering

source prec	target	order	#iter	#FMA	#mul	**#op**
$1/29$ (bit trick)	F32	4	1	4	3	**7**
2^{-12}	F32	1	1	1	3	**4**
2^{-14}	F32	1	1	1	3	**4**
$1/29$ (bit trick)	F64	3	2	6	6	**12**
2^{-12}	F64	4	1	4	3	**7**
2^{-14}	F64	3	1	3	3	**6**
2^{-23}	F64	2	1	2	3	**5**
2^{-28}	F64	1	1	1	3	**4**

All additions are fused

saves 2 multiplications and 1 subtraction (or 1 FMA and 1 multiplication). It is interesting not only because it requires fewer instructions, but also because it saves the need for two constants (0.5 and 3).

Algorithm 11: Householder for $1/\sqrt{x}$

input : x
input : $\hat{r}$ // estimation of $1/\sqrt{x}$
output: r // corrected estimation

$\alpha \leftarrow \hat{r} \cdot \hat{r} \cdot x$
$r \leftarrow \hat{r} \cdot (\frac{35}{16} - \alpha \cdot (\frac{35}{16} - \alpha \cdot (\frac{21}{16} - \frac{5}{16} \cdot \alpha)))$
// These fractions can exactly be represented as floating-point numbers and do not introduce any error

2) Householder: The Householder's method is a higher order generalization of the Newton-Raphson method. The speed of convergence can be chosen by choosing the order of the Householder's method:

- order 1: Newton-Raphson method: quadratic convergence
- order 2: Halley's method: cubic convergence
- order 3: quartic convergence
- $\cdots$

Sebah [25] explains how to find the iteration for a function f:

$$x_{n+1} = x_n - \frac{f_n}{f'_n}\left(1 + \frac{f_n f''_n}{2!\ {f'_n}^2} + \frac{{f_n}^2\left(3\ {f''_n}^2 - f'_n f^{(3)}_n\right)}{3!\ {f'_n}^4} + \cdots\right) \tag{3}$$

where $f^{(i)}_n = f^{(i)}(x_n)$

As for Newton-Raphson, we need to find the zero of $f(x) = \frac{1}{x^2} - a$. Stopping at order 3 gives the following iteration:

$$x_{n+1} = x_n\left(\frac{35}{16} - \frac{35}{16}\ {x_n}^2\, a + \frac{21}{16}\ {x_n}^4\, a^2 - \frac{5}{16}\ {x_n}^6\, a^3\right) \tag{4}$$

We can notice that in (4), in brackets is a polynomial of ${x_n}^2\, a$. This leads to:

$$\begin{aligned} \alpha_n &= {x_n}^2\, a \\ x_{n+1} &= x_n\left(\frac{35}{16} - \frac{35}{16}\ \alpha_n + \frac{21}{16}\ {\alpha_n}^2 - \frac{5}{16}\ {\alpha_n}^3\right) \end{aligned} \tag{5}$$

Horner scheme allows to compute a scalar polynomial with the least number of multiplication [26]. With Horner scheme, evaluating a n degree polynomial requires n multiplications and n additions. Moreover, these operations can be fused together. Thus, on CPUs with FMA, it can be computed with FMAs only. On current CPUs, FMA instructions are as fast as a single multiplication. This allows to write algorithm 11 which is the order 3 Householder's method for the square root reciprocal efficiently using only 3 multiplications and 3 FMAs. It is one multiplication less than using the Newton-Raphson method.

One can do the same calculations for other orders. It is then possible to see, for a given source accuracy and a given target precision, which order allows to compute full accuracy with a minimum number of operations. We computed the orders up to 5, and see which order requires the lowest number of operations. Results are reported in Table V.

V. Benchmarks

A. Benchmark protocol

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

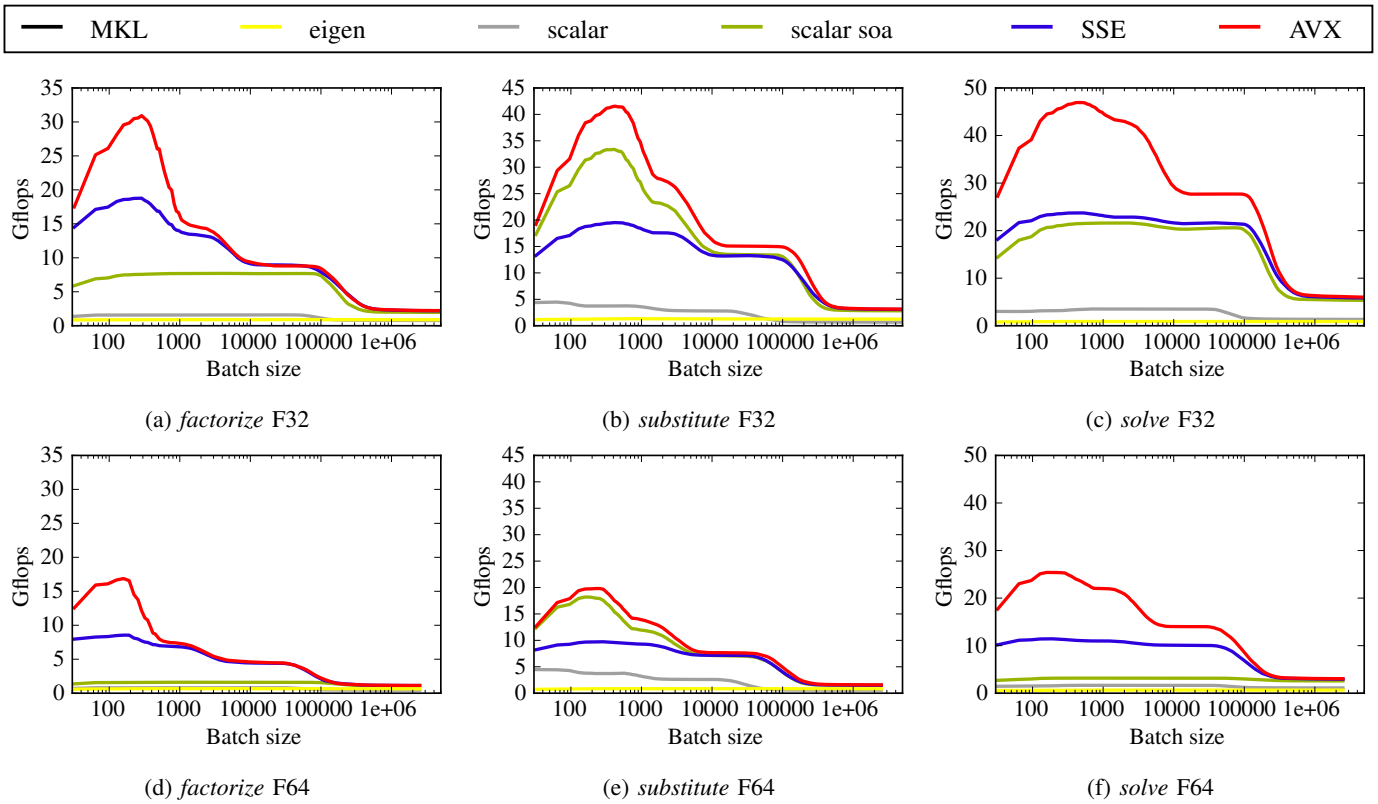
The algorithms were benchmarked on eight machines whose specifications are provided in Table VI.

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): $A \cdot X = B$

TABLE VI: Benchmarked machines and Stream TRIAD bandwidth

CPU	full name	cores/threads	cache (KB) per core *L1*	cache (KB) per core *L2*	cache (KB) per CPU *L3*	memory bandwidth (GB/s) 1 core	memory bandwidth (GB/s) 1 CPU	memory bandwidth (GB/s) 2 CPUs
HSW-i7	i7-4790	4/8	32	256	8192	7.9	7.9	–
SKL-i7	i7-6700K	4/8	32	256	8192	21	21	–
HSW Xeon	E5-2683 v3	2× 14/28	32	256	35840	11	39	77
KNC	7120P	61/244	32	512	–	5.3	300	–
KNL	7210	64/256	32	512	–	8.5	310	–
Power 8	8335-GCA Power 8	2× 8/64	64	512	65536	33	66	133
Rasp3	BCM2837 A53	4/4	32	512	–	2.0	2.2	–
TX1	jetson TX1 A57	4/4	32	512	–	7.1	9.5	–



(a) *factorize* F32 (b) *substitute* F32 (c) *solve* F32

(d) *factorize* F64 (e) *substitute* F64 (f) *solve* F64

Fig. 3: Impact of batch size on performance for 3×3 systems on HSW-i7, mono-core version

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

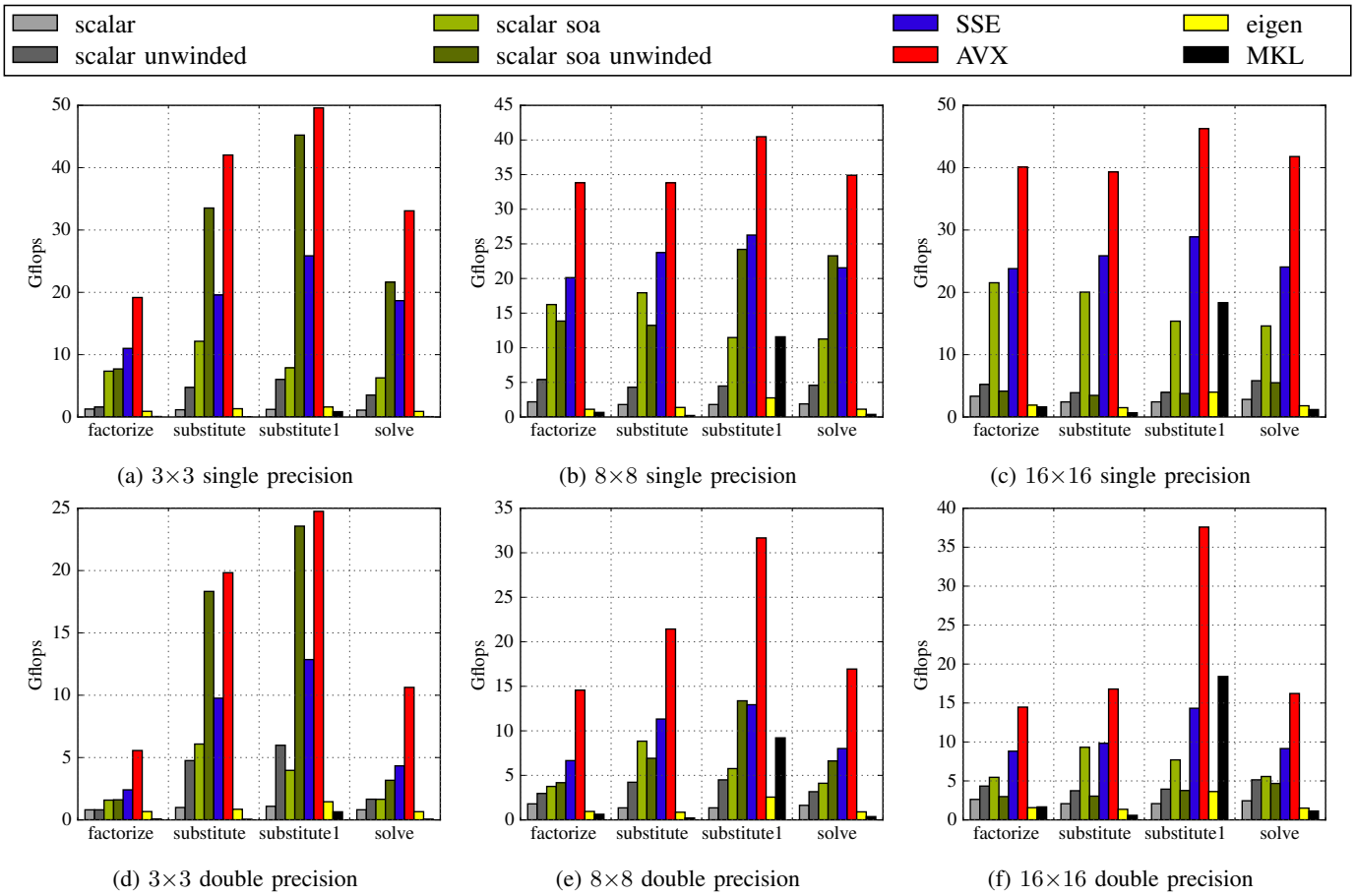
The time is measured with `_rdtsc()` which provides reliable time measures in cycles. Indeed, on current CPUs, the timestamp counter is normalized around the nominal frequency of the processor and is independent from any frequency changes.

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 for Intel architectures with Intel icc v17.0 with the following options: `-std=c99 -O3 -vec -ansi-alias` and for other architectures with gcc 6.2 with the following options: `-std=c99 -O3 -ffast-math -ftree-vectorize -fstrict-aliasing`.

The plots use the following conventions:

- Series labeled `scalar` are scalar written code. The *SoA* versions are vectorized by the compiler though.
- Series labeled `eigen` are eigen versions.
- Series labeled `SSE` are *SSE* code executed on the machine, even if it is an *AVX* machine.
- Series labeled `AVX` are *AVX* code executed on the machine.
- "`unwinded`" tag stands for inner loops



(a) 3×3 single precision (b) 8×8 single precision (c) 16×16 single precision

(d) 3×3 double precision (e) 8×8 double precision (f) 16×16 double precision

Fig. 4: Code performance of `factorize`, `substitute`, `substitute1` and *solve* in Gflops on HSW-i7, mono-core version

unwinded+scalarized (ie: fully unrolled).

- "`legacy`" tag stands for the version without the reciprocal storing (base version).
- "`fast`" tag stands for the use of fast square root reciprocal.
- "`fastest`" tag stands for the use of fast square root reciprocal estimation without any accuracy recovering.
- "`×k`" tags stand for the order of unrolling of the outer loop (unroll&jam)

B. Results

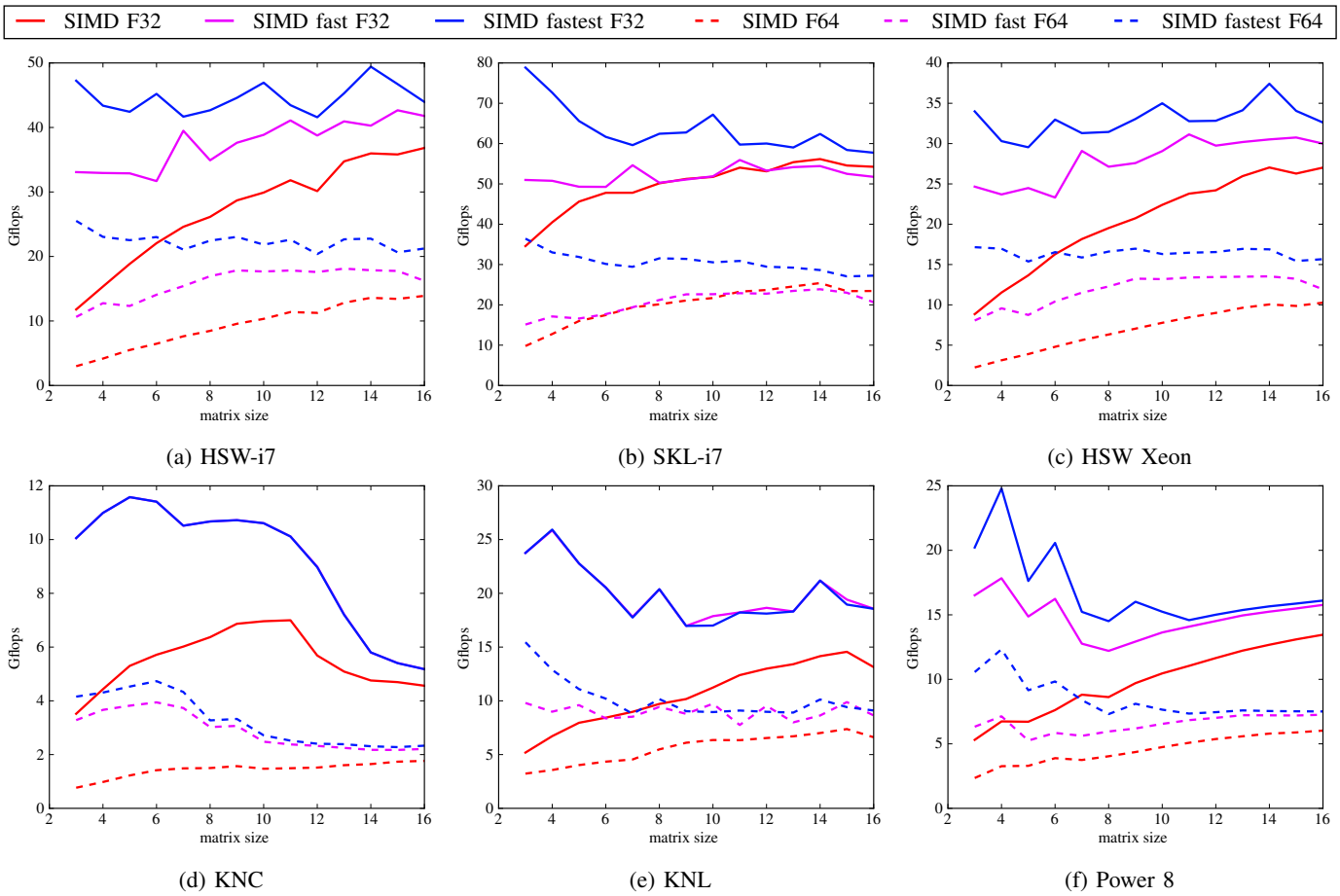
We focus our explanations on *solve* and the HSW-i7 machine. All the machines have similar behaviors unless explicitly specified otherwise. ARM code has not been compiled for ARM 64 bits (aarch64) and thus the double precision version is not present.

We first present the impact on performance of the batch size. We consider the performance of all the functions. After, we show the differences between single and double precision. Then we detail what transformations improve the performance and how. We exhibit with more details the impact of unrolling. And we show the scalability of our solution. Finally, we show summary results on all machines.

1) Batch size performance: Figure 3 shows important results for the understanding of our function performance. It shows the performance of *factorize*, *substitute* and *solve* on HSW-i7 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* chart (Figure 3a), one can notice 3 intervals of batch size for 3×3 matrices on HSW-i7:

- $[400, 1000]$: this is the *L1* cache overflow. As the *L1* cache is 32 KB, we cannot store data for more than 546 systems.
- $[3000, 8000]$: this is the *L2* cache overflow. As the *L2* cache is 256 KB, we cannot store data for more than 4,369 systems.
- $[10^5, 6{\cdot}10^5]$: this is the *L3* cache overflow. As the *L3* cache is 8 MB, we cannot store data for more than 139,810 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



(a) HSW-i7 (b) SKL-i7 (c) HSW Xeon

(d) KNC (e) KNL (f) Power 8

Fig. 5: Code performance in Gflops for single and double precision of the *solve* mono-core version

data fit within a cache, they may not be within it at the first execution: the cache is cold. But at the next execution, data will be in the cache: the cache warms up. 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.

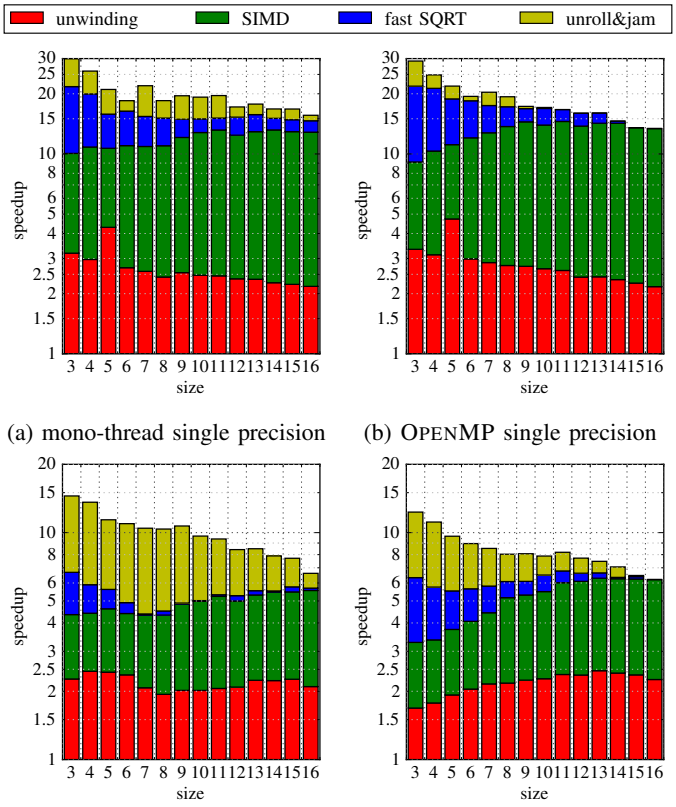
After the *L3* cache overflow, the best versions have almost the same performances: they are limited by the memory bandwidth. In this case, the bandwidth of the *factorize* function after the last cache overflow is about 7.9 GB/s, which is the bandwidth of the machine external memory.

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

2) Function performances: Figure 4 shows the performance of all 4 functions *factorize*, *substitute*, *substitute1* and *solve* in single and double precision for 3×3 and 16×16 matrices. When we look at Figure 4a and 4d, we can see that, for 3×3 matrices, the scalar *SoA* unwinded version performs very well on *substitute* and *substitute1* in both single and double precision, 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. In average, *AVX* version is twice faster than *SSE* version. Double precision has a similar shape than single precision.

The MKL is very slow. The reason is that it performs 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 not efficient for large matrices. However, for large matrices, these issues disappear and the MKL is extremely fast. Eigen has similar issues but is a bit faster on 3×3 matrices.

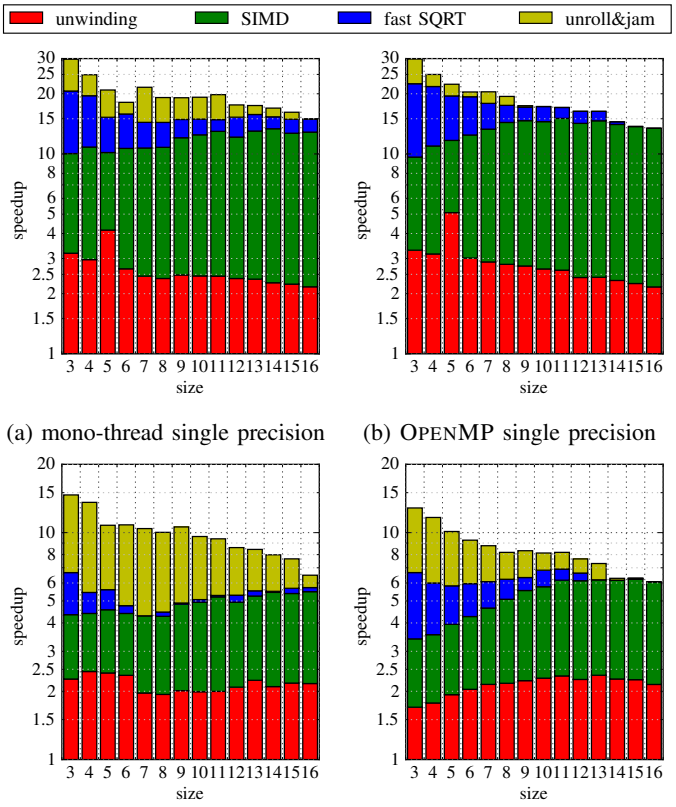
With 16×16 matrices, we can notice that all "slow" versions



(a) mono-thread single precision (b) OpenMP single precision

(c) mono-thread double precision (d) OpenMP double precision

Fig. 6: Speedups of the transformations for *solve* on HSW-i7 in single and double precision mono-core and multi-core



(a) mono-thread single precision (b) OpenMP single precision

(c) mono-thread double precision (d) OpenMP double precision

Fig. 7: Speedups of the transformations for *solve* on HSW Xeon in single and double precision mono-core and multi-core

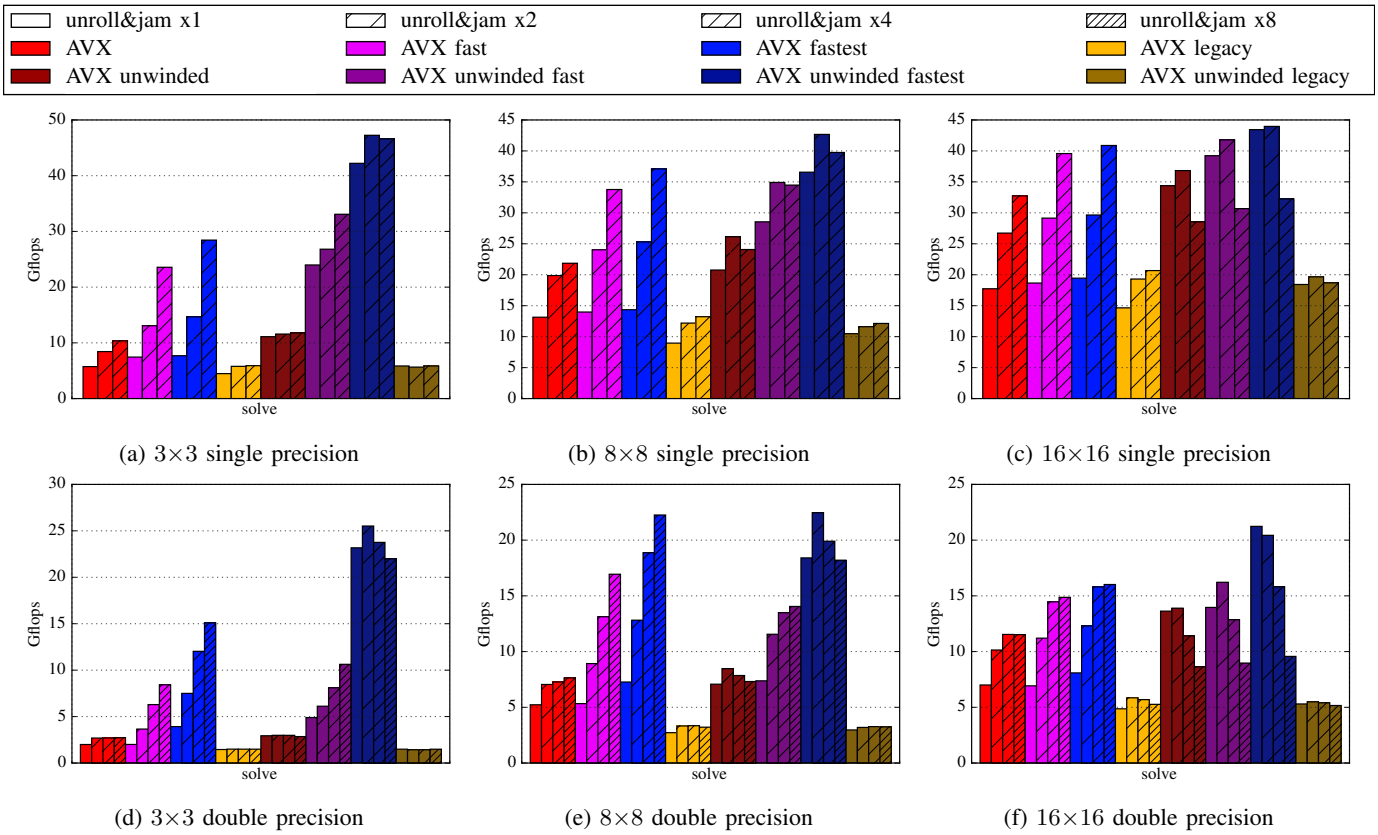
are a faster. We can also see that the MKL is much better on *substitute1*: it is able to deal with batches, and overhead is reduced. However, the scalar *SoA* unwinded version is much slower: at this point, the compiler does not vectorize this code anymore.

3) `float` vs `double`: When we compare 32-bit single precision (`float`) performance with 64-bit double precision (`double`) performance in Figure 3 (3a, 3b, 3c vs 3d, 3e, 3f), we can see that the plots are similar. There is two main differences. First, the `double` version is slower than the `float` version. It can be easily explained by *SIMD* cardinal: a `float` *SIMD* instruction is able to compute twice more data as the `double` one in the same time or less. Quantitative comparison will be addressed later in the paper. The second difference is about cache overflow. On the `double` version, cache overflows happen twice earlier: the size of `double` is twice the size of `float`, but the cache remains the same size, so the number of `double` that can be in the cache is half.

On plots of Figure 5, we can see that the speedup of `float` over `double` is higher than $\times 2$: between $\times 3$ and $\times 4$ for both "non-fast" and "fast" on HSW-i7 (5a), SKL-i7 (5b), HSW Xeon (5c) and KNC (5d). A factor of 2 is explained by the cardinal of *SIMD* registers. The extra speedup depends on which version is considered. For "non-fast" versions, IEEE 754 divisions and square roots are used. These instructions are slower for `double`s than for `float`s (Table II) and compute half the number of elements. The time to compute a square root or a division per element is then more than twice the time in `float`.

For fast versions, no square root nor division instruction is used. However, a fast square root reciprocal estimate is computed, and then the accuracy is recovered with the Newton-Rahpson method or the Householder's method. These methods require more iterations in `double` than in `float` because there is more precision to recover. So there is more computation to do in double precision. This also explains why the speedup "fast" over "non-fast" is higher in single precision than in double precision.

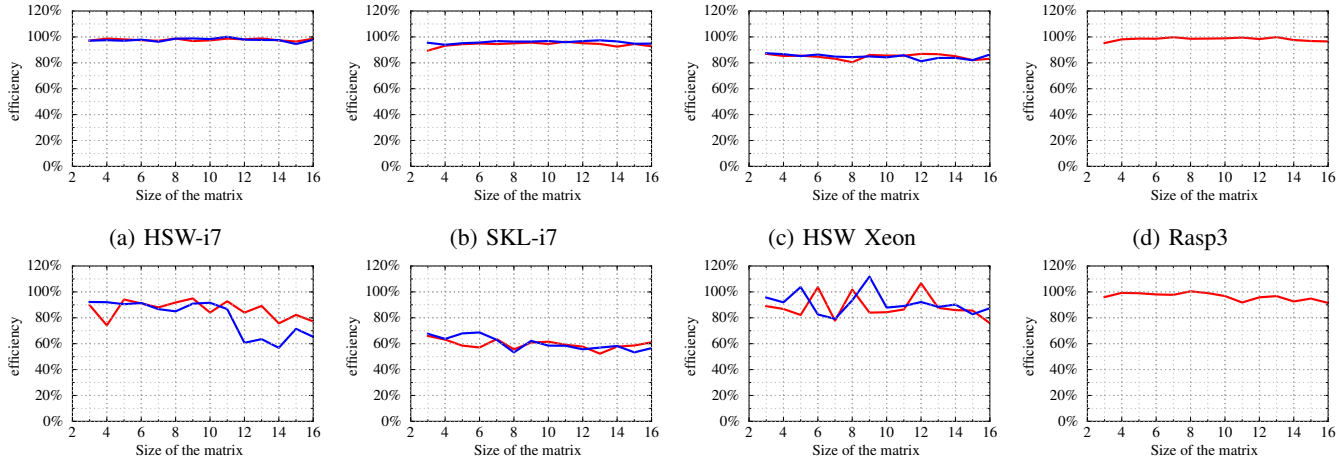
On KNC and KNL in single precision (Figure 5d and 5e), "fast" and "fastest" versions are completely identical (same code). These architectures have a square root reciprocal instruction that give full accuracy in single precision (Table III), so there is no need for a Newton-Raphson iteration.



(a) 3×3 single precision (b) 8×8 single precision (c) 16×16 single precision

(d) 3×3 double precision (e) 8×8 double precision (f) 16×16 double precision

Fig. 8: Performance of loop transforms and square root transforms for the *AVX* version of *solve* on HSW-i7 in Gflops



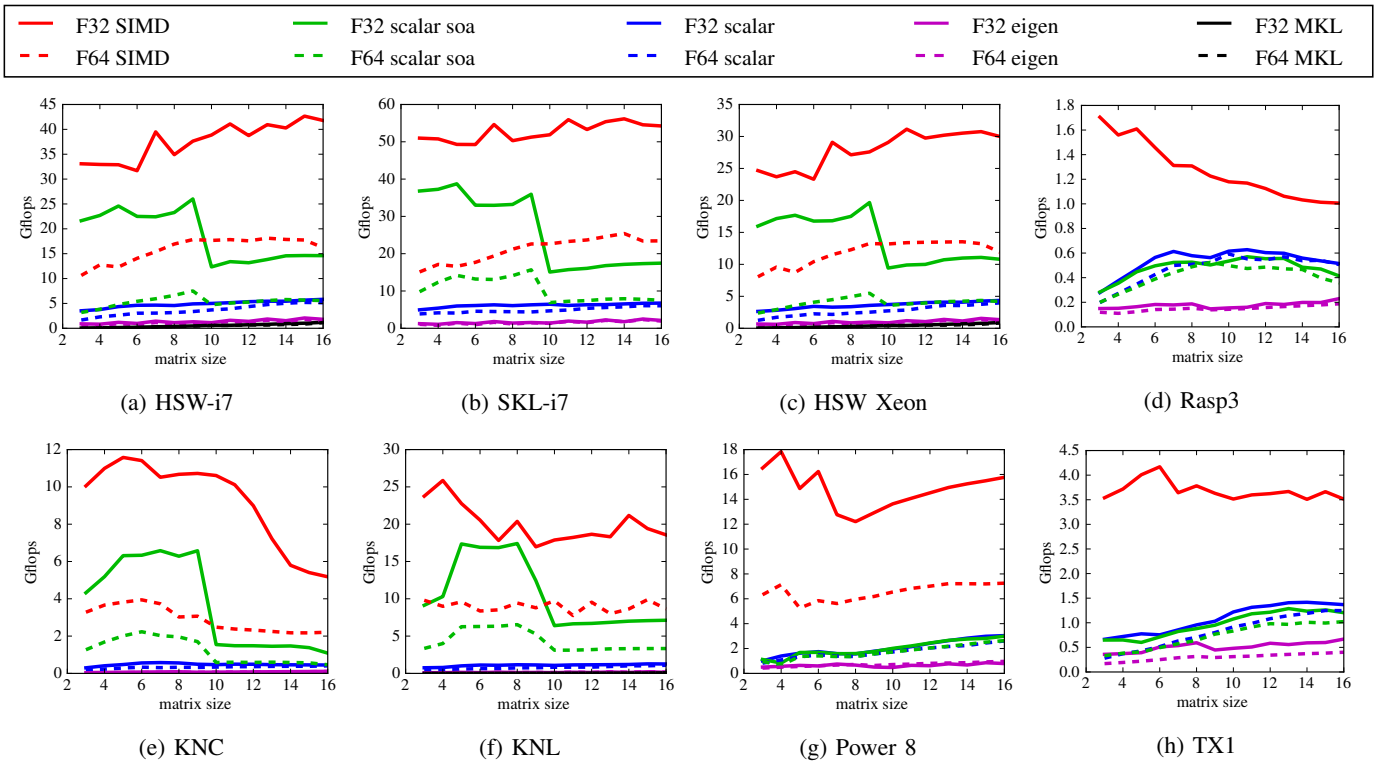
(a) HSW-i7 (b) SKL-i7 (c) HSW Xeon (d) Rasp3

(e) KNC (f) KNL (g) Power 8 (h) TX1

Fig. 9: multithreading efficiency of *SIMD solve*: single precision in red and double precision in blue

4) Incremental speedup: Figure 6 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.

If we look at the speedups on HSW-i7 mono-thread single precision (Figure 6a), we can see that unwinding the inner loops improves the performance well: from ×2 to ×3. The impact of unwinding decreases when the size of the matrix increases: the register pressure is higher. Looking at the assembly, we can actually see that the compiler generates spill code for large matrices. Spill code consists in moving values from register to memory to free a register, and moving back



(a) HSW-i7 (b) SKL-i7 (c) HSW Xeon (d) Rasp3

(e) KNC (f) KNL (g) Power 8 (h) TX1

Fig. 10: Mono-core performance of *solve* in Gflops on tested machines

when the value is needed again.

SIMD gives a sub-linear speedup: from $\times 3.2$ to $\times 6$. In fact, *SIMD* instructions cannot be fully efficient on this function without fast square root (see subsection IV-B). 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.

If we look at the multithreaded version (Figure 6b), results are similar. We can notice that the speedup of fast SQRT + unroll&jam is similar on both single thread and multithread charts, but the fast square root gives more speedup. This is especially visible on the double precision version. This is due to the hyperthreading that has an effect similar to unroll&jam allowing to use free functional units of a core for another thread by interleaving instructions within the processor pipeline.

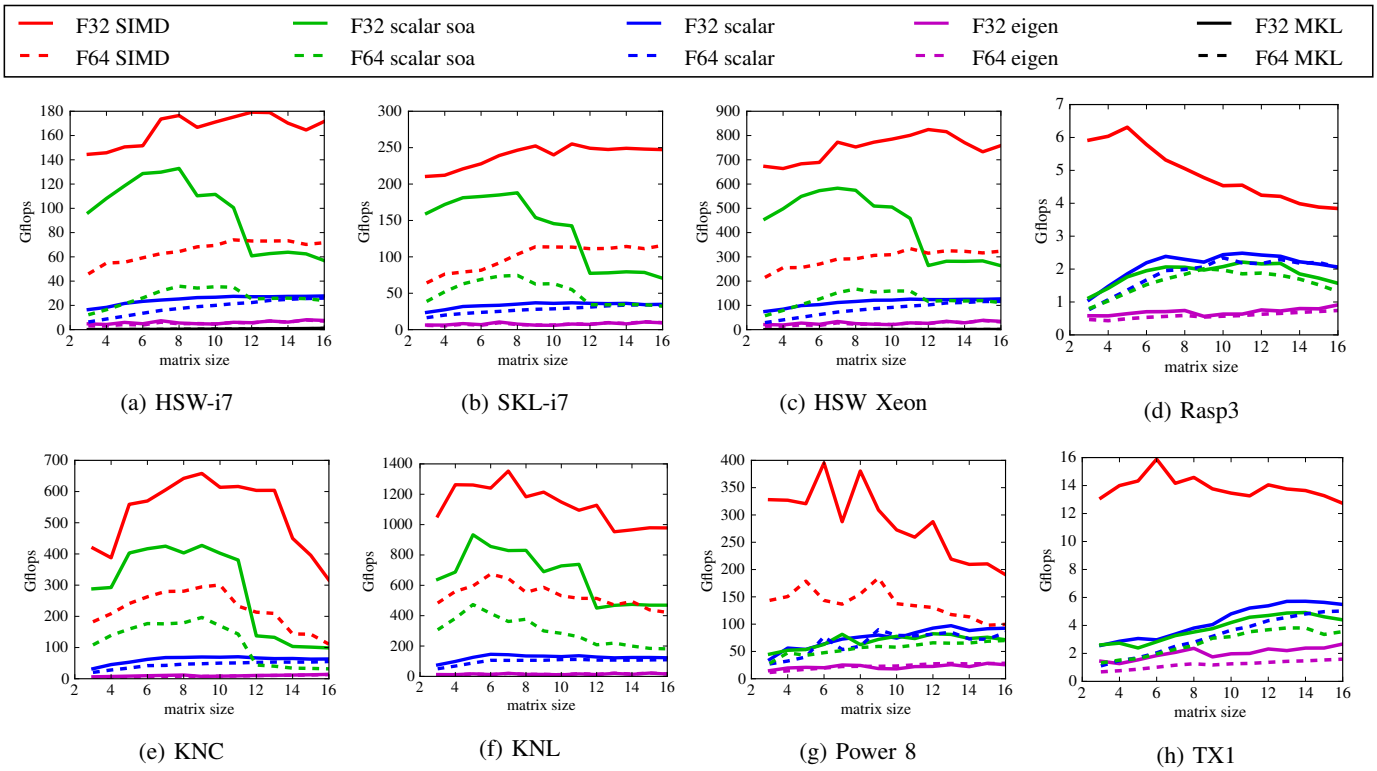
The doube precision versions are not exactly the same (Figure 6c). The speedup of the unwinding/scalarization transformation does not decrease with the size of the matrix. In double precision, the required bandwidth is higher, so saving memory loads and stores has more impact. One can expect this speedup to decrease with even larger matrices. Another difference is the impact of the fast square root + unroll&jam. On HSW-i7, this set of transformations gives a higher speedup on double precision than in single precision. This is due to the latency of the square root and division instructions, and the throughput of the fast square root reciprocal. On this machine, square root and division instructions have a high latency and without unroll&jam, the performance of this very code is limited by the instruction latencies. With unroll&jam, the performance is limited by the instruction throughputs and fast square root reciprocal computation is highly pipelined. The double precision square root and division instructions have a much higher latency on this machine, while the fast square root reciprocal throughput in double precision is still good. On SKL-i7, this effect is not visible as the square root and division instructions have a lower latency than on HSW-i7.

If we look at HSW Xeon (Figure 7) we see similar results.

5) Impact of unrolling: Figure 8 shows the performance of *solve* for different *AVX* versions.

Without any unrolling, all versions except "legacy" have similar performance: performance seems to be limited by the latency between data-dependent instructions. Unwinding can help Out-of-Order engine and thus reduce data-dependency.

For 3×3 matrices (Figure 8a and 8d), the performance of the "non-fast" and "legacy" versions are limited by the square



(a) HSW-i7 (b) SKL-i7 (c) HSW Xeon (d) Rasp3

(e) KNC (f) KNL (g) Power 8 (h) TX1

Fig. 11: Multi-core performance of *solve* in Gflops on tested machines

root and division instruction throughput. The performance has reached a limit and cannot be improved further this limitation, even with unrolling: both unwinding and unroll&jam are inefficient in this case. The "legacy" version is more limited because it requires more divisions. These versions are even more limited in double precision as square root and division instructions are even slower.

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. This transformation reduces pipeline stalls between data-dependent instructions (subsection III-C). 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).

The unwinded "fastest" versions give an important benefit especially in double precision. By removing the accuracy recovering instructions, we save a lot of instructions (IV-C, Accuracy recovering). As the number of instructions to recover double precision is higher compared to single precision, the speedup of "fastest" over "fast" is higher in double precision.

For 16×16 matrices (Figure 8c and 8f), the performance of all versions are leveled. The transformations we have done are good for small matrices, but become less efficient for bigger matrices.

- unwinding: register pressure becomes higher (for a $n\times n$ matrix, the code needs $\mathcal{O}(n^3)$ registers).
- unroll&jam: it allows to hide latencies, but with larger matrices, computations are more independent from each other.
- fast square root: the proportion of square roots and divisions decreases with the size of the matrix: these operations are diluted among the others.

For such large matrices, unroll&jam slows down the code when it is already unwinded because of the register pressure.

6) multithread scaling: Figure 9 shows the efficiency of the multithreading for the best *SIMD* version of *solve*. The efficiency is defined as the speedup of the multi-core code over the single core (hyperthreaded) code divided by the number of cores.

The scaling is strong (between 80% and 100%) for all multi-core machines: HSW-i7, SKL-i7, HSW Xeon, TX1 and Power 8. On manycore machines (KNC and KNL), the scaling is lower. On KNC, the scaling is strong for small matrices and gets lower for larger matrices, especially in double precision. On KNL, the scaling is lower: $\sim 60\%$ for all sizes. A memory bottleneck is suspected for both manycore architectures.

7) Summary: Figure 10 and Figure 11 show the performance of our best *SIMD* version against scalar versions and library versions (Eigen and MKL) for all architecture in both mono-core and OpenMP. The MKL is not present on the multi-core

TABLE VII: Speedups of the best *SIMD* version of *solve* over the scalar *AoS* version on all machines

Machine	mono-core		multi-core	
	F32	F64	F32	F64
HSW-i7	$\times 16 - \times 30$	$\times 6.6 - \times 15$	$\times 13 - \times 29$	$\times 6.2 - \times 12$
SKL-i7	$\times 16 - \times 39$	$\times 7.6 - \times 15$	$\times 15 - \times 33$	$\times 7.8 - \times 12$
HSW Xeon	$\times 14 - \times 28$	$\times 6.1 - \times 14$	$\times 12 - \times 30$	$\times 6.6 - \times 13$
KNC	$\times 51 - \times 350$	$\times 24 - \times 130$	$\times 27 - \times 120$	$\times 11 - \times 60$
KNL	$\times 73 - \times 420$	$\times 35 - \times 170$	$\times 33 - \times 150$	$\times 15 - \times 68$
Power 8	$\times 12 - \times 38$	$\times 5.9 - \times 16$	$\times 3.8 - \times 27$	$\times 2.1 - \times 10$
Rasp3	$\times 3.6 - \times 15$	N/A	$\times 3.5 - \times 14$	N/A
TX1	$\times 5.4 - \times 12$	N/A	$\times 4.8 - \times 13$	N/A

results as its heuristic limits multithreading for tiny problems. So this is not possible to compare our implementation with the MKL on multithreaded code. Both Eigen and the MKL are slower than our scalar *AoS* code.

We can see that for Intel architecture, scalar *SoA* is good for small matrices, but becomes slow after 9×9 matrices in mono-core. This is due to the compiler icc: it is able to vectorize the unwinded code up to 9×9. For larger matrices, it stops vectorizing. The threshold is after for the multithreaded versions, probably due to a change in the compiler heuristic. On other machines, gcc was used: gcc is unable to vectorize our scalar code, and there is no way to enforce it to vectorize, unlike icc. Writing *SIMD* code is mandatory to achieve efficient code as compilers are not always able to vectorize scalar code, even while enforcing them.

The speedup of the best version compared to the scalar *AoS* version for all tested architectures is in Table VII.

We achieve a high overall speedup for 3×3 up to 16×16 matrices compared to the basic scalar version. On HSW Xeon, we reach a $\times28$ speedup on single precision and a $\times14$ speedup on double precision. On Rasp3, we reach a $\times15$ speedup on single precision. And on Power 8, we reach a $\times38$ speedup on single precision and a $\times16$ speedup on double precision. The code scales also very well with a mutlithread efficiency above 80% on most of the machines.

Conclusion

In this paper, we have presented an efficient *SIMD* implementation for tiny matrices ($\leqslant 16\times16$) of the Cholesky algorithm, because in some fields they are very used and because State-of-the-Art libraries are inefficient for such tiny matrices.

Moreover, on some architectures like ARM Cortex or IBM Power, the existing optimizing compilers are unable to vectorize this kind of code. On other architectures like x86, some compilers are able to vectorize but not efficiently and not for all sizes. Hand-written *SIMD* code is thus mandatory to fully benefit from architecture.

To reach a high level of performance, the proposed implementation combines low level transformations (loop unrolling and loop unwinding), hardware optimizations (*SIMD* and multi-core) and High Level Transforms (fast square root and memory layout). We achieve a high overall speedup outperforming existing codes for *SIMD* CPU architectures on tiny matrices on both single precision and double precision: a speedup of $\times30$ on a high-end Intel Xeon workstation, $\times15$ on a ARM Cortex embedded processor and $\times38$ on a IBM Power 8 HPC cluster node.

References

[1] MKL, "Intel(R) math kernel library." https://software.intel.com/en-us/intel-mkl.

[2] S. Tomov, R. Nath, P. Du, and J. Dongarra, "Magma, matrix algebra on gpu and multicore architectures." http://icl.cs.utk.edu/magma/.

[3] G. Guennebaud, B. Jacob, *et al.*, "Eigen v3." http://eigen.tuxfamily.org/, 2016.

[4] A. R. M. y Terán, L. Lacassagne, A. H. Zahraee, and M. Gouiffes, "Real-time covariance tracking algorithm for embedded systems," in *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pp. 104–111, IEEE, 2013.

[5] 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*, vol. 262, no. 2, pp. 444–450, 1987.

[6] 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*, pp. 495–501, IEEE, 1997.

[7] 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*, pp. 353–370, Springer, 2011.

[8] 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*, vol. 2015, pp. 1–14, Jan. 2015.

[9] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. Dongarra, "High-performance matrix-matrix multiplications of very small matrices," in *European Conference on Parallel Processing*, pp. 659–671, Springer International Publishing, 2016.

[10] 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*, pp. 432–440, IEEE, 2014.

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

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

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

[14] N. J. Higham, "Cholesky factorization," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 1, no. 2, pp. 251–254, 2009.

[15] 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*, pp. 157–160, IEEE, 2014.

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

[17] 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)*, pp. 49–56, 2014.

[18] JINJA2, "Python template engine." http://jinja.pocoo.org/.

[19] I. Masliah, M. Baboulin, and J. Falcou, "Metaprogramming dense linear algebra solvers applications to multi and many-core architectures," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 3, pp. 69–76, IEEE, 2015.

[20] J. Abel, K. Balasubramanian, M. Bargeron, T. Craver, and M. Phlipot, "Applications tuning for streaming SIMD extensions," *Intel Technology Journal*, vol. 2, 1999.

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

[22] 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.
[23] P. Soderquist and M. Leeser, "Area and performance tradeoffs in floating-point divide and square-root implementations," *ACM Comput. Surv.*, vol. 28, pp. 518–564, Sept. 1996.
[24] C. Lomont, "Fast inverse square root," tech. rep., 2003.
[25] P. Sebah and X. Gourdon, "Newton's method and high order iterations," tech. rep., 2001.
[26] V. Y. Pan, "Methods of computing values of polynomials," *Russian Mathematical Surveys*, vol. 21, no. 1, pp. 105–136, 1966.