Small SIMD Matrices for CERN High Throughput Computing

Florian Lemaitre, Benjamin Couturier, Lionel Lacassagne

Small SIMD Matrices for CERN High Throughput Computing

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

***Abstract*—System tracking is an old problem and has been heavily optimized throughout the past. However, in High Energy Physics, many small systems are tracked in real-time using Kalman filtering and no implementation satisfying those constraints currently exists. In this paper, we present a code generator used to speed up Cholesky Factorization and Kalman Filter for small matrices. The generator is easy to use and produces portable and heavily optimized code. We focus on current SIMD architectures (SSE, AVX, AVX512, Neon, SVE, Altivec and VSX). Our Cholesky factorization outperforms any existing libraries: from $\times 3$ to $\times 10$ faster than MKL. The Kalman Filter is also faster than existing implementations, and achieves $4 \cdot 10^9$ iter/s on a $2{\times}24$C Intel Xeon.**

I. Introduction

The goal of the paper is to present a code generator and optimizations to get a fast reconstruction of a system trajectory (tracking) using Kalman filtering for SIMD multi-core architectures, for which it exists no efficient implementation. The constraints are strong: few milliseconds to track thousands of particles. Right now, the choice was to focus on general-purpose processors (GPP) as SIMD extensions are present in every system (so all CERN researcher could benefit of it). GPUs were not selected when the work started in 2015 as the transfer time (through PCI) between the host and the GPU was longer than the amount of time allocated to the computation. With the rise of the last generation of GPU connected to a CPU with a high bandwidth bus, it becomes worth evaluating them.

Even though optimizing Kalman filter tracking is an old problem [21], existing implementations are not efficient for many small systems.

The code generator uses the template engine Jinja2 [13] and implements high level and low level optimizations. It is used to produce a fast Cholesky factorization routine and a fast Kalman filter in C. The generated code is completely generic and can be used with any system. It also supports many SIMD architectures: SSE, AVX, AVX512, Neon, SVE, Altivec and VSX. In order to have a representative Kalman filter and validate its implementation, a basic $4{\times}4$ system was selected. Depending on the experiment the matrix size can change. Some specific variants can also exist: $5{\times}5$ systems for High Energy Physics [9], using three steps: forward, backward, smoother.

This work will be used in the next upgrade of the LHCb experiment to achieve real-time event reconstruction.

Algorithm 1: Cholesky system solving $A \cdot X = R$

```
// Factorization
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         $L(i,j) \leftarrow s/L(j,j)$

// Forward substitution
11 for $i = 0 : n-1$ do
12     $s \leftarrow R(i)$
13     for $j = 0 : i-1$ do
14         $s \leftarrow s - L(i,j) \cdot Y(j)$
15     $Y(i) \leftarrow s/L(i,i)$

// Backward substitution
16 for $i = n-1 : 0$ do
17     $s \leftarrow Y(i)$
18     for $j = i+1 : n-1$ do
19         $s \leftarrow s - L(j,i) \cdot X(j)$
20     $X(i) \leftarrow s/L(i,i)$
```

TABLE I: Arithmetic Intensity (AI)

version	flop	load + store	AI
classic	$\frac{1}{6}\left(2n^3 + 15n^2 + 7n\right)$	$\frac{1}{6}\left(2n^3 + 12n^2 + 40n\right)$	~ 1
scalarized		$\frac{1}{2}\left(n^2 + 6n\right)$	$\sim 2n/3$

In this paper, we will first present Cholesky Factorization, the optimizations we applied to it, and their performance impact. Then, we will present Kalman Filter, its optimizations and their performance impact.

II. Cholesky factorization

A. Algorithm

Cholesky Factorization (also known as Cholesky Decomposition) is a linear algebra algorithm used to express a symmetric positive-definite matrix as the product of a triangular matrix with its transposed matrix: $A = L \cdot L^T$. It can be combined with forward and backward substitutions to solve a linear system (algorithm 1).

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 [11], [12]. 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 use 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.

With small matrices, parallelization is not efficient as there is no long dimension. Therefore, matrices are grouped by batches in order to efficiently parallelize along this new dimension. 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].

B. Transformations

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

- High Level Transforms: memory layout [1] and fast square root (the latter is detailed in II-C),
- loop transforms (loop unwinding [16] and unroll&jam),
- Architectural transforms: SIMDization.

1) Memory Layout Transform: The memory layout transform is the first transform to address as the other ones rely on it. The default memory layout in C is Array of Structure (*AoS*), but is not suited for SIMD. In order to enable SIMD, the layout should be modified into Structure of Arrays (*SoA*). A hybrid memory layout (*AoSoA*) is preferred to avoid systematic cache evictions.

The alignment of the data is also crucial. Aligned memory allocations 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 line 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.

2) Loop unwinding: Loop unwinding is the special case of loop unrolling where the loop is entirely unrolled. it has several advantages, especially for small matrices:

- it avoids branching,
- it allows to keep temporaries 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. This leads to algorithm 2 and reduces the amount of memory accesses (Table I).

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

3) Loop Unroll & Jam: 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 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 throughputof the instruction and not by its latency.

Current processors are Out-of-Order. But they are limited by the size of their rescheduling window. In order to help

Algorithm 2: Cholesky system solving $A \cdot X = R$ unwound and scalarized for 4×4 matrices

```
   // Load $A$ into registers
1  $a_{00} \leftarrow A(0,0)$
2  $a_{10} \leftarrow A(1,0)$   $a_{11} \leftarrow A(1,1)$
3  $a_{20} \leftarrow A(2,0)$   $a_{21} \leftarrow A(2,1)$   $a_{22} \leftarrow A(2,2)$
4  $a_{30} \leftarrow A(3,0)$   $a_{31} \leftarrow A(3,1)$   $a_{32} \leftarrow A(3,2)$   $a_{33} \leftarrow A(3,3)$
   // Load $R$ into registers
5  $r_0 \leftarrow R(0)$   $r_1 \leftarrow R(1)$   $r_2 \leftarrow R(2)$   $r_3 \leftarrow R(3)$
   // Factorize $A$
6  $l_{00} \leftarrow \sqrt{a_{00}}$
7  $l_{10} \leftarrow a_{10}/l_{00}$
8  $l_{20} \leftarrow a_{20}/l_{00}$
9  $l_{30} \leftarrow a_{30}/l_{00}$
10 $l_{11} \leftarrow \sqrt{a_{11} - {l_{10}}^2}$
11 $l_{21} \leftarrow (a_{21} - l_{20} \cdot l_{10})/l_{11}$
12 $l_{31} \leftarrow (a_{31} - l_{30} \cdot l_{10})/l_{11}$
13 $l_{22} \leftarrow \sqrt{a_{22} - {l_{20}}^2 - {l_{21}}^2}$
14 $l_{32} \leftarrow (a_{32} - l_{30} \cdot l_{20} - l_{31} \cdot l_{21})/l_{22}$
15 $l_{33} \leftarrow \sqrt{a_{33} - {l_{30}}^2 - {l_{31}}^2 - {l_{32}}^2}$
   // Forward substitution
16 $y_0 \leftarrow r_0/l_{00}$
17 $y_1 \leftarrow (r_1 - l_{10} \cdot y_0)/l_{11}$
18 $y_2 \leftarrow (r_2 - l_{20} \cdot y_0 - l_{21} \cdot y_1)/l_{22}$
19 $y_3 \leftarrow (r_3 - l_{30} \cdot y_0 - l_{31} \cdot y_1 - l_{32} \cdot y_1)/l_{33}$
   // Backward substitution
20 $x_3 \leftarrow y_3/l_{33}$
21 $x_2 \leftarrow (y_2 - l_{32} \cdot x_3)/l_{22}$
22 $x_1 \leftarrow (y_1 - l_{21} \cdot x_2 - l_{31} \cdot x_3)/l_{11}$
23 $x_0 \leftarrow (y_0 - l_{10} \cdot x_1 - l_{20} \cdot x_2 - l_{30} \cdot x_3)/l_{00}$
   // Store $X$ into memory
24 $X(3) \leftarrow x_3$   $X(2) \leftarrow x_2$   $X(1) \leftarrow x_1$   $X(0) \leftarrow x_0$
```

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. Its efficiency is limited by the throughput of the unrolled loop instructions and the register pressure.

C. Precision and Accuracy

Cholesky Factorization 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 (see [7]) and usually not fully pipelined. Thus, square roots and divisions limit the overall Cholesky throughput.

It is possible in hardware to compute them faster with less accuracy [23]. 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).

1) 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 $x \cdot L(i,i)^{-1}$. The algorithm then needs only n divisions.

2) Fast square root reciprocal estimation: The algorithm performs a division by a square root and therefore needs to

Listing 1: Simple C loop

```
for (int i = 0; i < 4; i++) {
  s = B[i] + C[i];
  A[i] = s / 2;
}
```

Listing 2: Simple Jinja loop

```
{% for i in range(4) %}
s{{i}}   = B[{{i}}] + C[{{i}}];
A[{{i}}] = s{{i}} / 2;
{% endfor %}
```

compute $f(x) = 1/\sqrt{x}$. There are some ways to compute an estimation of this function depending on the 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 any 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 and gives an estimation with 12-bit accuracy. Unlike regular square root and division, this instruction is fully pipelined ($throughput = 1$) and thus avoids pipeline stall.

3) Accuracy recovering: Depending on the application, the previous techniques might not be accurate enough. The accuracy recovering (if needed) can be done with Newton-Raphson method or Householder's. All current SIMD architectures have FMAs instruction to apply those methods quickly. See [17] for more details.

D. Code generation

In order to help writing many different versions of the code, we used Jinja2 [13]: a template engine in Python. Using this tool, we can easily implement unrolling (both unwinding and unroll&jam) and intrinsics code. The syntax uses custom tags/tokens that control what is being output. As it is text substitution, it is possible to manipulate new identifiers.

The generated code features all transformations and all sizes from 3×3 up to 12×12 for all the architectures supported and all SIMD wrappers. There is no actual limit for the unrolled size, but the bigger the matrices, the longer the compilation. This could be replaced by a C++ template metaprogram like in [19]. The use of Jinja2 instead of more common metaprogramming methods allows us to have full access and control over the generated code. In some applicative domains, it is crucial to have access to the source code before the compilation in order to quickly track bugs.

1) unrolling: Unwinding can be done in Jinja by replacing the C `for`-loop (Listing 1) into a Jinja `for`-loop (Listing 2). The output of the template is the C code the compiler will see (Listing 3).

Unroll&Jam uses a Jinja filter: a filter is a section that is interpreted by Jinja as usual, but the output is then passed to a function to transform it directly in Python. The `unrollNjam` filter duplicates lines with the symbol `@`, and replace the `@` by 0, 1, 2... The template code in Listing 4 generates the code in Listing 5.

Listing 3: Simple Jinja loop output

```
s0   = B[0] + C[0];
A[0] = s0 / 2;
s1   = B[1] + C[1];
A[1] = s1 / 2;
s2   = B[2] + C[2];
A[2] = s2 / 2;
s3   = B[3] + C[3];
A[3] = s3 / 2;
```

2) SIMD: The SIMD generation is handled via a custom C-like preprocessor written in Python. The interface consists in custom Python objects accessible from Jinja.

When a Python macro is used within Jinja (Listing 6), it is replaced by a unique name that is detected by our preprocessor. It then acts like a regular C-preprocessor and replaces the macro call by its definition from the Python class (Listing 7).

It is important to have a preprocessor as all the architecture intrinsics differ not only by their name, but also by their signature. The Altivec code (Listing 8) looks completely different from for SSE despite being generated from the same template (with VSX, the output would involve `vec_mul` instead of `vec_madd`). This tool can also be used to generate code for C++ SIMD wrappers.

3) SIMD wrappers: In order to see if it is worth writing intrinsics, SIMD wrappers have been integrated into the code and compared to the intrinsics and scalar code. The following libraries have been tested: Boost.SIMD [6], libsimdpp [18], MIPP [3], UME::SIMD [14], vcl [8].

Eigen has also been tested but is unable to compile Cholesky Factorization when the element type is an array. It would have been possible to write manually the factorization array element type, but this would defeat the whole point of Eigen.

More libraries and tools exist like CilkPlus, Cyme, ispc [22], Sierra or VC [15]. CilkPlus, Cyme and Sierra appear not to be maintained anymore. VC and ispc did not fit into our test code base without a lot of efforts, thus were not tested.

Listing 4: Unroll&Jam in Jinja

```
{% filter unrollNjam(range(4)) %}
s@   = B[@] + C[@];
A[@] = s@ / 2;
{% endfilter %}
```

Listing 5: Unroll&Jam output

```
s0   = B[0] + C[0];
s1   = B[1] + C[1];
s2   = B[2] + C[2];
s3   = B[3] + C[3];
A[0] = s0 / 2;
A[1] = s1 / 2;
A[2] = s2 / 2;
A[3] = s3 / 2;
```

Listing 6: Macro usage

```
{{vec}} h, a, b;
h = {{vec(0.5)}};
a = {{vec.load}}(&A[i]);
b = {{vec.mul}}(a, h);
{{vec.store}}(&B[i], b);
```

Listing 7: Macro output (SSE)

```
__m128 h, a, b;
h = _mm_set1_ps(0.5f);
a = _mm_load_ps(&A[i]);
b = _mm_mul_ps(a, h);
_mm_store_ps(&B[i], b);
```

SIMD wrappers in C++ are much longer to compile than plain C with intrinsics. The biggest file took more than 30 hours and required more than 10 GB of memory to compile. Thus, it was decided to stop the generation of unrolled code for matrices bigger than 12×12.

E. Benchmarks

1) Benchmark protocol: In order to evaluate the impact of the transforms, we used exhaustive benchmarks.

The algorithms were benchmarked on six machines whose specifications are provided in Table II.

On x86, the code has been compiled with Intel icc v18.0 with the following options: `-std=c99 -O3 -vec -ansi-alias`. The time is measured in cycles with `_rdtsc()`.

On other architectures, gcc 7.2 has been used with the following options: `-std=c99 -O3 -ffast-math -fstrict-aliasing`. Time is measured with `clock_gettime(CLOCK_MONOTONIC, ...)`.

In all the cases, the code is run multiple times with multiple batch sizes, and the best time is kept.

The plots use the following conventions:

- `scalar`: scalar code. The *SoA* versions are vectorized by the compiler though.
- `SIMD`: SIMD intrinsics code executed on the machine.
- `unwind`: inner loops unwound+scalarized (ie: fully unrolled).
- `legacy`: no reciprocal storing (base version).
- `fast`: use of fast square root reciprocal estimation.
- `fastest`: "`fast`" without any accuracy recovering.
- `×k`: order of the outer loop unrolling (unroll&jam)

We focus our explanations on the HSW machine and single precision as the accuracy is enough. See [17] for the analysis of the double precision computation. All the machines have similar behaviors unless explicitly specified otherwise.

We first present the impact of the transforms on performance. Then, we compare our best version written in intrinsics with SIMD wrappers and MKL [20]. Finally, we show the performance on multiple machines.

Listing 8: Macro output (Altivec)

```
vector float h, a, b;
h = ((vector float){0.5f, 0.5f, 0.5f, 0.5f});
a = vec_ld(0, &A[i]);
b = vec_madd(a, h, (vector float)vec_splat_u32(0));
vec_st(b, 0, &B[i]);
```

2) Incremental speedup: Figure 1 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 significant.

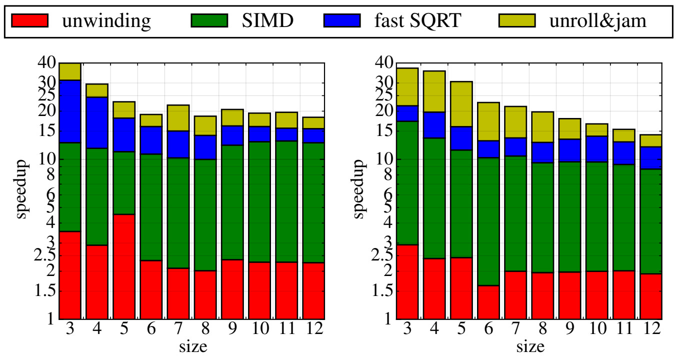
If we look at the speedups on HSW (Figure 1a), we can see that unwinding the inner loops improves the performance well: from ×2 to ×3. Unwinding impact decreases when the matrix size increases: the register pressure is higher.

SIMD gives a sub-linear speedup: from ×3.2 to ×6. In fact, SIMD instructions cannot be fully efficient on this function without fast square root (see subsubsection II-C2). With further analysis, we can see that the speedup of SIMD + fast square root is almost constant around ×6. The impact of the fast square root decreases as their number becomes negligible compared to the other floating-point operations. For small matrices, unroll&jam allows to get the last part of the expected SIMD speedup. SIMD + fast square root + unroll&jam: from ×6.5 to ×9. Unroll&jam loses its efficiency for larger matrices: the register pressure is higher.

Speedups on Power8 are similar: Figure 1b.

3) Impact of unrolling: Figure 2 shows the performance 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 reduces data-dependencies.

The performance of the "non-fast" and "legacy" versions are limited by the square 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 as it requires more divisions.



(a) HSW (b) Power8

Fig. 1: Speedups of the transformations for Cholesky

TABLE II: Benchmarked machines

CPU	full name	ISA	frequency (GHz)	cores/threads	SIMD width	#FMA	SIMD SP parallelism (FLOP/cycle)	cache (KB) per core *L1*	cache (KB) per core *L2*	cache (KB) per CPU *L3*
HSW	E5-2683 v3 [a]	AVX2	2.0	2× 14/28	256	2	32	32	256	35840
i9	i9-7900X[a]	AVX512	3.3	10/20	512	2	64	32	1024	14080
SKX	Platinum 8168 [a]	AVX512	2.7	2× 24/48	512	2	64	32	1024	33792
EPYC	EPYC 7351P [b]	AVX2	2.4	16/32	256	1	16	32	512	65536
A72	Cortex A72 [c]	Neon	2.4	2× 32/32	128	1	8	32	256	32768
Power8	Power 8 Turismo [d]	VSX	3.0	4/32	128	2	16	64	512	8192

[a] Intel [b] AMD [c] ARM [d] IBM

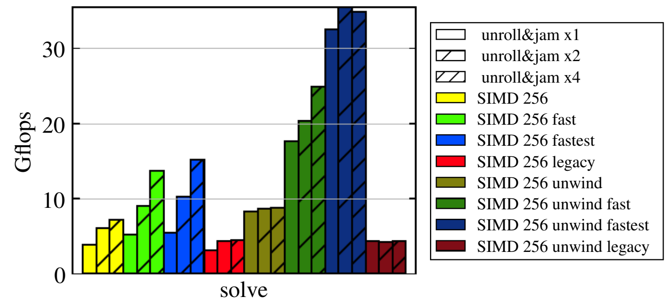


Fig. 2: Performance of loop and square root transforms for the AVX 3×3 version of Cholesky on HSW

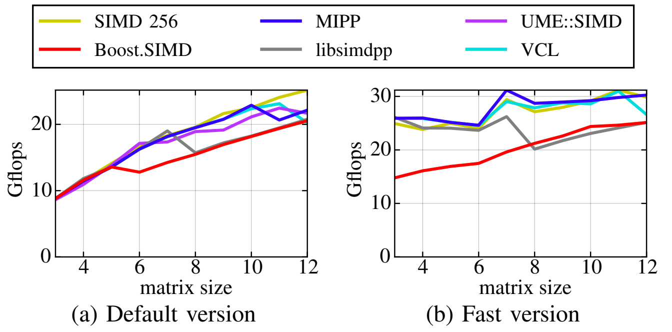


Fig. 3: Performance comparison between intrinsics code and SIMD wrappers for Cholesky on HSW

For "fast" versions, both unrolling are efficient. Unroll&jam achieves a ×3 speedup on regular code and ×1.5 speedup with unwinding. This transformation reduces pipeline stalls between data-dependent instructions (subsubsection II-B3). We can see that unroll&jam is less efficient when the code is already unwound but keeps improving the performance. Register pressure is higher when unrolling (unwinding or unroll&jam).

The "unwind+fastest" versions give an important benefit. By removing the accuracy recovering instructions, we save many instructions (II-C3, Accuracy recovering).

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

4) SIMD wrappers: Figure 3 shows the performance of SIMD wrappers compared to the intrinsics version. Optimizations not related to SIMD are applied the same way on all versions. With the default version, all the wrappers seem to have the performance until a point depending on the wrapper. The drop in performance is a bug of the compiler that stops inlining the wrapper functions when the outer function is too big (unwinding+unroll&jam).

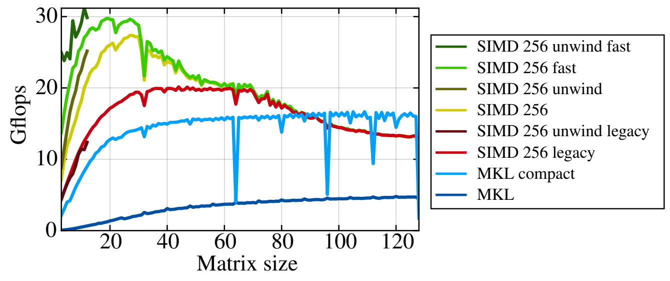


Fig. 4: Performance comparison between intrinsics code and MKL for Cholesky on HSW

With the "fast" version, most wrappers have similar performance in single precision. However, UME::SIMD does not implement the square root reciprocal approximation (despite being part of the interface). Moreover, only Boost.SIMD supports the fast square root in double precision. In that case, Boost.SIMD is a bit slower than the intrinsics code.

5) Comparison with MKL: The version 2018 now supports the *SoA* memory layout. It is designated by *compact* within the documentation. Figure 4 shows the performance comparison between our implementation and MKL.

The *compact* layout improved the performance for small matrices, compared to the old functions. However, it is still slower than our version for matrices smaller than 90×90.

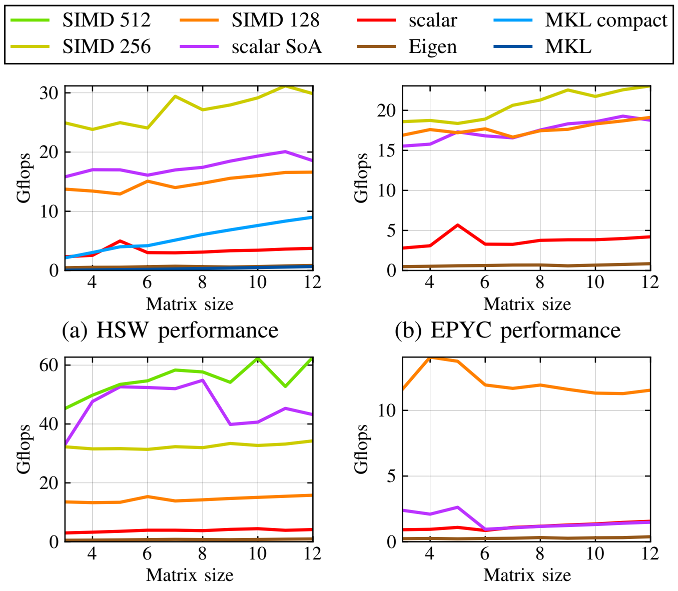
First, MKL does not store the reciprocal and has to compute actual divisions during both factorization and substitution. This can be compared to our "legacy" version.

Then, it uses a recursive algorithm for the substitution that has some overhead.

6) Summary: Figure 5 shows the performance of our best SIMD version against scalar versions and libraries (Eigen and MKL) for HSW, SKX, EPYC and Power8. Due to licensing limitations, MKL has only been tested on HSW.

On aarch64, gcc has a performance bug[1] where the instrinsic `vmlsq_f32(a,b,c)` $= a - b \cdot c$ is compiled into two instructions instead of one. This bug also affects the instrinsic `vfmsq_f32`. As Cholesky Factorization mainly uses the latter intrinsic, the performance obtained on this machine has no meaning and was not considered here.

[1] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=82074



(a) HSW performance (b) EPYC performance

(c) SKX performance (d) Power8 performance

Fig. 5: Performance of Cholesky on HSW, SKX, EPYC and Power8 machines, mono-core

Both Eigen and the classic routines of MKL are slower than our scalar *AoS* code and are barely visible on the plots. The "compact" routines of MKL are faster, but still much slower than the SIMD version.

On SKX, the scalar *SoA* performance drops from 9×9 matrices. This is due to the compiler icc that stops vectorizing the unwound scalar code from this point.

On all tested machines, the scaling is strong with a parallel efficiency[2] above 80%.

III. Kalman filter

A. Kalman Filter algorithm

Kalman Filter is a well-known algorithm to estimate the state of a system from noisy and/or incomplete measurements. It is commonly used in High Energy Physics and Computer Vision as a tracking algorithm (reconstruct the trajectory). It is also used for positioning systems like GPS.

Kalman filtering involves few matrix multiplications and a matrix inversion that can be done with Cholesky Factorization (see algorithm 3).

[2]The parallel efficiency is defined as the speedup of the multi-core code over the single core code divided by the number of cores.

Algorithm 3: Kalman Filter

```
in/out : $x$, $P$      // state, covariance
input  : $u$, $z$      // control, measure
input  : $A$, $B$, $Q$, $H$, $R$   // Parameters of the Kalman filter

// Predict
1 $x' \leftarrow A\,x + B\,u$
2 $P' \leftarrow A\,P\,A^\intercal + Q$
// Innovation
3 $\tilde{y} \leftarrow z - H\,x'$
4 $S \leftarrow H\,P'\,H^\intercal + R$
// Kalman Gain
5 $K \leftarrow P'\,H^\intercal S^{-1}$
// Update
6 $x \leftarrow x' + K\,\tilde{y}$
7 $P \leftarrow (I - K\,H)\,P'$
```

We focus on 4×4 Kalman filtering in order to validate the implementation while keeping a representative filter. However, the code is not limited to 4×4 systems and actually supports all sizes. The filtered system is an inertial point in 2D with the following state: $(x, y, \dot{x}, \dot{y})$.

B. Transformations

All transformations applied to Cholesky Factorization have been tested on Kalman Filter. A few other optimizations have been implemented and tested: algebraic optimizations and memory access optimizations.

1) Algebraic optimizations: When optimizing an algorithm like Kalman filtering, one can try to optimize the mathematical operations.

The first thing to consider is avoiding the recomputation of temporaries that are used several times. For the Kalman filter from algorithm 3, it is possible to keep the temporary product $P'H^\intercal$ (line 4) to compute K (line 5).

It is also possible to keep S in its factorized form and expand the expression of K in the expression of x and P: algorithm 4. This ends up being less arithmetic operations as long as matrix-vector products are preferred over matrix-matrix products.

Algorithm 4: Kalman filter Optimized

```
in/out : $x$, $P$      // state, covariance
input  : $u$, $z$      // control, measure
input  : $A$, $B$, $Q$, $H$, $R$   // Parameters of the Kalman filter

// Predict
1 $x' \leftarrow A\,x + B\,u$
2 $P' \leftarrow A\,P\,A^\intercal + Q$
// Innovation
3 $\tilde{y} \leftarrow z - H\,x'$
4 $\Gamma \leftarrow P'\,H^\intercal$
5 $S \leftarrow H\,\Gamma + R$
// "Kalman Gain"
6 $L \leftarrow \mathrm{cholesky}(S)$
7 $M \leftarrow L^{-1}\,\Gamma^\intercal$
// Update
8 $x \leftarrow x' + \Gamma\,L^{-1\intercal}\,L^{-1}\tilde{y}$
9 $P \leftarrow P' - M^\intercal\,M$
```

2) Memory access of symmetric matrices: One can save many memory loads and stores by accessing only half of the symmetric matrices. Indeed, those matrices are used a lot within Kalman filtering for covariance matrices.

When the matrices are in *AoS*, accessing only half of a symmetric matrix decreases a lot the vectorization efficiency, especially with small matrices. Indeed, the pattern to access the near-diagonal elements is not regular.

However, when matrices are in *SoA*, there is no such penalty as we always load entire registers. Therefore, the vectorization efficiency is the same as for square matrices, except with fewer operations and memory accesses.

C. Benchmarks

1) Benchmark protocol: We use essentially the same protocol to test our Kalman filter as for the Cholesky factorization. The Kalman filter considered has a 4-dimensional state space $(x, y, \dot{x}, \dot{y})$. Many of these systems are tracked together. The time is measured per iteration. The plots use the same conventions as for Cholesky, plus these extra:

- `v1`: classic version of the algorithm (algorithm 3)
- `v2`: optimized version of the algorithm (algorithm 4)
- `triangle`: only half of symmetric matrices is accessed

2) Incremental speedup: Incremental speedups are reported on Figure 6. Like with Cholesky, the speedup comes mainly from unwinding and *SoA* memory layout that enables vectorization. The mathematical optimizations (`v2`+`triangle`) give a total extra speedup about $+40\%$.

Unlike with Cholesky, the fast square and unroll&jam give no benefit except on Power8. Indeed, the proportion of square roots and division is much lower on Kalman. Moreover, the operations are more independent from each other (more intrinsic parallelism). Therefore, unroll&jam is not efficient here. However, it is still interesting without unwinding.

The last thing to notice is that writing SIMD intrinsics does not improve the performance, except on Power8 where gcc struggles to optimize for the Power architecture.

3) Overall performance: The machines available for testing at CERN are very different: two high-end bi-socket (Intel, ARM) and two mono-socket (AMD, Power). So in order to provide fair comparisons, we have normalized the results to focus on transform speedups, and not the raw performance.

Looking at Figure 7, it appears clearly that it is not worth writing SIMD as compilers are able to vectorize the code. We still have to supply `#pragma omp simd` to ensure the vectorization. Otherwise, compiler heuristics would have stopped vectorizing.

Doing that, the compiler is even able to provide slightly better code than SIMD code. The instruction scheduling and register allocation might be involved.

On A72, the SIMD code is even slower than vectorized because of the gcc bug.

Like with the Cholesky factorization, the scaling is strong with a parallel efficiency above 80%.

4) State-of-the-art: As previously said, each experiment implements some specific version of Kalman filtering, direct comparison cannot be done. Indeed, the problem dimensionality is different and the steps are different. Moreover, each step of the filter for HEP is lighter than the full Kalman filtering: no control vector, one-dimension measurement space.

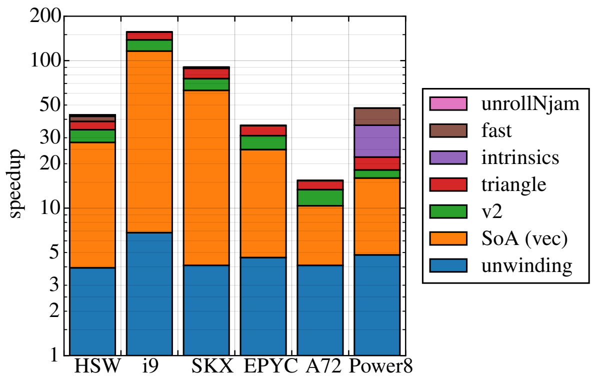


Fig. 6: Incremental speedup of the Kalman filter

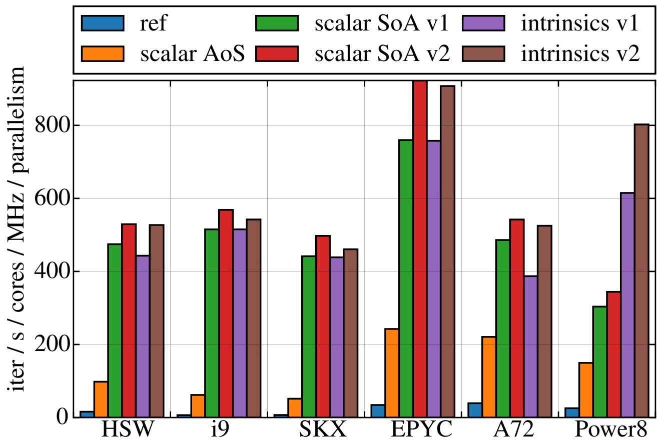


Fig. 7: Overall normalized performance of the Kalman filter

TABLE III: Rough comparison with State-of-the-Art Kalman filters for HEP

Implementation	steps	cycle/iter
our code (4×4)	FWD	44
our code (5×5)	FWD	74
CMS (5×5)	FWD+BWD+smooth	520
CBM (5×5)	FWD+BWD+smooth	550
LHCb (5×5)	FWD+BWD+smooth	1440

Timing of other implementations have been estimated from their article

Nevertheless, the performance of the SIMD implementations for CMS [4], CBM [10] and LHCb [2] is between 500 and 1500 cycle/iter (all steps). Our 4×4 implementation achieves 44 cycle/iter (Table III). This is an order of magnitude faster than existing implementations.

As a matter of fact, the SKX machine reaches an overall performance of $4 \cdot 10^9$ iter/s.

Conclusion

In this paper, we have presented a code generator used to create an efficient and portable SIMD implementation of Cholesky Factorization for small matrices ($\leqslant 12\times12$) and Kalman Filter for 4×4 systems. The generated code supports many SIMD architectures, and is AVX512/SVE ready. Being completely general, it can be used with any system and is not limited to 4×4 systems.

Our Cholesky factorization outperforms any existing libraries. Even if there are some improvements with MKL, we are still $\times3$ up to $\times10$ faster on small matrices.

Our Kalman filter implementation is not directly comparable to the State-of-the-Art because of its general form, but appears to be one order of magnitude faster. With this, we are able to reach $4 \cdot 10^9$ iter/s on a high-end Intel Xeon 2×24C.

To reach such a high level of performance, the proposed implementation combines high level transforms (fast square root and memory layout), low level transforms (loop unrolling and loop unwinding), hardware optimizations (SIMD and OpenMP multithreading) and linear algebra optimizations. The code was automatically generated using Jinja2 to provide strong optimizations with simple source code. SIMD wrappers allow to write portable SIMD code, but require extra optimizations handled by our code generator.

With GPUs directly connected to the main memory, the transfer bandwidth is much higher; thus, it would be worth considering GPUs for future work.

References

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

[2] Cámpora Pérez, D. H. LHCb Kalman Filter cross architecture studies. *Journal of Physics: Conference Series 898*, 3 (2017), 032052.

[3] Cassagne, A., Le Gal, B., Leroux, C., Aumage, O., and Barthou, D. An efficient, portable and generic library for successive cancellation decoding of polar codes. In *International Workshop on Languages and Compilers for Parallel Computing* (2015), Springer, pp. 303–317.

[4] Cerati, G., Elmer, P., Krutelyov, S., Lantz, S., Lefebvre, M., McDermott, K., Riley, D., Tadel, M., Wittich, P., Wurthwein, F., and Yagil, A. Kalman filter tracking on parallel architectures. *Journal of Physics: Conference Series 898*, 4 (2017), 042051.

[5] Dong, T., Haidar, A., Luszczek, P., Harris, J. A., Tomov, S., and Dongarra, J. 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* (2014), IEEE, pp. 157–160.

[6] Estérie, P., Falcou, J., Gaunard, M., and Lapresté, J.-T. Boost.SIMD: Generic programming for portable SIMDization. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing* (New York, NY, USA, 2014), WPMVP '14, ACM, pp. 1–8.

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

[8] Fog, A. C++ vector class library. http://agner.org/optimize/vectorclass.zip, 2017. accessed version: 2017-07-27.

[9] Frühwirth, R. 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 (1987), 444–450.

[10] Gorbunov, S., Kebschull, U., Kisel, I., Lindenstruth, V., and Müller, W. Fast SIMDized Kalman filter based track fit. *Computer Physics Communications 178*, 5 (2008), 374–383.

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

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

[13] JINJA2. Python template engine. http://jinja.pocoo.org/.

[14] Karpiński, P., and McDonald, J. A high-performance portable abstract interface for explicit SIMD vectorization. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores* (New York, NY, USA, 2017), PMAM'17, ACM, pp. 21–28.

[15] Kretz, M., and Lindenstruth, V. Vc: A C++ library for explicit vectorization. *Software: Practice and Experience 42*, 11 (2012), 1409–1430.

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

[17] Lemaitre, F., Couturier, B., and Lacassagne, L. Cholesky factorization on simd multi-core architectures. *Journal of Systems Architecture 79*, C (Sept. 2017), 1–15.

[18] libsimdpp. Header-only zero-overhead c++ wrapper for simd intrinsics of multiple instruction sets. https://github.com/p12tic/libsimdpp, 2017. commit: 7c2b867.

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

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

[21] Palis, M. A., and Krecker, D. K. Parallel Kalman filtering on the connection machine. In *Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the* (1990), IEEE, pp. 55–58.

[22] Pharr, M., and Mark, W. R. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012* (2012), IEEE, pp. 1–13.

[23] Soderquist, P., and Leeser, M. Area and performance tradeoffs in floating-point divide and square-root implementations. *ACM Comput. Surv. 28*, 3 (Sept. 1996), 518–564.