MIPP: a Portable C++ SIMD Wrapper and its use for Error Correction Coding in 5G Standard

Adrien Cassagne*†, Olivier Aumage*, Denis Barthou*, Camille Leroux† and Christophe Jégo†
* Inria, Bordeaux Institute of Technology, LaBRI/CNRS, Bordeaux, France
† IMS/CNRS, Bordeaux, France

ABSTRACT

Error correction code (ECC) processing has so far been performed on dedicated hardware for previous generations of mobile communication standards, to meet latency and bandwidth constraints. As the 5G mobile standard, and its associated channel coding algorithms, are now being specified, modern CPUs are progressing to the point where software channel decoders can viably be contemplated. A key aspect in reaching this transition point is to get the most of CPUs SIMD units on the decoding algorithms being pondered for 5G mobile standards. The nature and diversity of such algorithms requires highly versatile programming tools. This paper demonstrates the virtues and versatility of our *MIPP* SIMD wrapper in implementing a high performance portfolio of key ECC decoding algorithms.

KEYWORDS

SIMD, wrapper, C++, channel code, SSE, AVX, AVX-512, NEON

ACM Reference Format:
Adrien Cassagne*†, Olivier Aumage*, Denis Barthou*, Camille Leroux† and Christophe Jégo†. 2018. MIPP: a Portable C++ SIMD Wrapper and its use for Error Correction Coding in 5G Standard. In *WPMVP'18 : Workshop on Programming Models for SIMD/Vector Processing, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3178433.3178435

1 INTRODUCTION

The standardization of the fifth generation of mobile network (5G) is currently under discussion [9]. As any communication standard, it will define the protocols to be used on the different abstraction layers of a mobile network. Regarding the lower layers (PHY and MAC layers), the standard specifies several methods and algorithms to be applied on the digital information to be transmitted. Among them, error correction codes and their associated encoding/decoding algorithms represent what is called *channel coding*. The purpose of this data processing is to add redundancy to the message on the transmitter side. On the receiver side, the decoder exploits redundancy to detect and correct the potential errors in the received message. Among all the PHY and MAC layer algorithms, channel coding is the most computationally intensive. This is especially true for the decoding algorithms on the receiver side. Moreover, channel coding processing must be performed under stringent timing constraints: low latency and high throughput. This explains why, in the previous generations of mobile communication standards (4G, 3G, ...), channel coding was performed on dedicated hardware. With recent multicore general purpose processors, it seems possible to execute channel decoding algorithms on programmable hardware under real time constraints. Furthermore, in the future 5G standard, this migration of PHY and MAC layers processing from dedicated hardware to programmable processors is pushed to such an extent that the overall network could actually be completely virtualized. In this new paradigm, denoted as Cloud-RAN (Radio Access Network), instead of being executed on dedicated hardware close to the antennas, all the digital data processing would be performed on centralized large scale programmable processors arrays. This would allow a dynamic balancing of the computational effort within the network [6, 20, 22, 26, 27].

In this context of network virtualization, recent articles have proposed several optimized software decoders, corresponding to different channel codes : LDPC codes [16, 17], polar codes [4, 5, 11, 28], turbo codes [3, 30, 31]. All of these works show the possibility to reach a good level of *performance* by making extensive use of SIMD (Single Instruction Multiple Data) units. This is often achieved at the price of a reduced *flexibility*, by resorting to specific intrinsics, or by making assumptions on the data types. However, these decoders should be implemented in a single source code, in which the following parameters could be changed at runtime: the channel code type, the decoding algorithm, the number of decoding iterations, the data format, etc. Another important aspect is the *portability* of the source code on different hardwares (Intel x86, Xeon KNL and ARM) and the possibility to use different instruction sets (`SSE`, `AVX`, `AVX-512`, `NEON`). These three constraints (performance, flexibility, portability) push towards the use of a SIMD library that helps in the abstraction of the SIMD instruction sets, while still allowing a fine grain tuning of performance.

We propose in this paper a new `C++` SIMD wrapper, covering the needs in terms of expressiveness and of performance for the channel codes. Our contributions are:

- A portable and high performance `C++` SIMD wrapper called `MIPP`, for `SSE`, `AVX`, `AVX-512` and `NEON` instruction sets;
- An implementation of several channel codes with this wrapper. We present the main advantages of `MIPP` in this context;
- A comparison with other state-of-the-art SIMD wrappers on a Mandelbrot code, demonstrating that the code based on `MIPP` has similar performance as hand-written intrinsics.

`MIPP` programming model is not too far from intrinsics, allowing a good control on performance, but still provides an abstraction on

WPMVP'18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5646-6/18/02…$15.00
https://doi.org/10.1145/3178433.3178435

the basic types used in vectors (ranging from double to byte) and complex operations (parametric reductions, log, exponential, ...).

2 PROPOSED WRAPPER

The MyIntrinsics++ library (MIPP) is a portable wrapper for SIMD intrinsics written in the C++ language. It relies on C++ compile-time template specialization techniques to replace supported generic functions with inline calls to their intrinsics counterpart, for a given instruction set. While MIPP is mostly written in C++98, it still requires C++11-compliant compiler due to the use of convenient features such as the `auto` and `using` keywords. MIPP is Open-source (under the MIT license) and the full code is available on GitHub[1].

MIPP provides two application programming interface levels. The *Low Level Interface* (low) implements a basic, thin abstraction layer on top of the intrinsics. The *Medium Level Interface* (med.), built on top of MIPP low, abstracts away more details to lessen the effort from the application programmer by relying on object encapsulation and operator overloading.

2.1 Low Level Interface

MIPP low is built around a unified `mipp::reg` type that abstracts vector registers. The vector register type represents hardware registers independently of the data type of the vector elements. MIPP uses the longest native vector length available on the architecture. This design choice preserves programmer flexibility, for instance in situations such as mixing fixed-point and floating-point operations. MIPP also defines a *mask* register type `mipp::msk`, which either directly maps to real hardware masks on instruction sets that support it (such as AVX-512), or to simple vector registers otherwise.

MIPP low then defines a set of functions working with `mipp::reg` and `mipp::msk`, organized into eight families: memory accesses, shuffles, bitwise boolean arithmetic, integer operations, float. operations, mathematical functions, reductions, and mask operations.

In the AVX-512 instruction set, one *regular* vector operation plus one masking operation can be performed in one CPU clock cycle. For instance, the following instruction performs "`m ? a+b : src`", an addition and a masking operation:

```
__m512 _mm512_mask_add_ps(__m512 src, __mmask16 m,
                          __m512 a,   __m512    b);
```

MIPP natively supports such operations with the `mipp::mask` function. The previous example becomes in MIPP:

```
mipp::mask<float,mipp::add<float>>(m, src, a, b);
```

For instruction sets without masking support, the `mipp::mask` call is expanded as an operation and a `blend` instead.

2.2 Medium Level Interface

The MIPP Medium Level Interface (MIPP med.) provides additional expressiveness to the programmer. `mipp::reg` and `mipp::msk` basic types are encapsulated in `mipp::Reg<T>` and `mipp::Msk<N>` objects, respectively. The `T` and `N` template parameters correspond to the type and the number of elements inside the vector register and the mask register, respectively. One can notice that in these register objects are typed, unlike the MIPP low register basic type. It avoids to write the type when a MIPP function is called. The function type can then be directly selected from the parameter type. Listing 1 illustrates the template-based encapsulation, which enables MIPP to override common arithmetic and comparison operators.

Listing 1: Medium Level Interface encapsulation.

```
template <typename T>
class Reg {
  mipp::reg r; // the register type from MIPP low
  Reg(const T *ptr) : r(mipp::load<T>(ptr)) {}
  inline Reg<T> add(const Reg<T> r) const {
    return mipp::add<T>(r,r.r);
  }
  inline Reg<T> operator+(const Reg<T> r) const {
    return this->add(r);
  } /* ... */
};
```

MIPP med. also simplifies register loading and initialization operations. The constructor of the `mipp::Reg` object will call the `mipp::load` function automatically. Thus, a load in MIPP low:

```
mipp::reg a = mipp::load<float>(aligned_ptr);
```

can be simplified into:

```
mipp::Reg<float> a = aligned_ptr;
```

with MIPP med. level. An initializer list can be used with a MIPP med. vector register:

```
mipp::Reg<float> a = {1.f, 2.f, 3.f, 4.f};
```

Likewise, a scalar assigned to a vector sets all elements to this value.

2.3 Implementation Details

MIPP targets SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2, FMA3, KNCI, AVX-512F and AVX-512BW instruction sets on x86 and related architectures, as well as NEON, NEONv2, NEON64 and NEON64v2 on ARM. It can easily be extended to other instruction sets.

MIPP selects the most recent instruction set available at compile time. For instance, a code compiled with the `-march=avx` flag of the GNU GCC compiler uses AVX instructions even if the architecture supports SSE as well. The vector register size is determined by the instruction set and the data type. A dedicated function returns the number of elements in a MIPP register:

```
constexpr int n = mipp::nElmtsPerRegister<T>();
```

A shortened version is also defined as: `mipp::N<T>()`. Whenever vectorization takes place in loops, MIPP's philosophy is to change the stride of the loop from one to the size of registers. The stride can be statically determined with the `mipp::N<T>()` function. If the loop size is not a multiple of the registers size, 1) a sequential tail loop can be implemented to compute the remaining elements, 2) the padding technique can be implemented to force the loop size to be a multiple of the vector registers.

When the instruction set cannot be determined, MIPP med. falls back on sequential instructions. In this case, MIPP does not use any intrinsic anymore. However, the compiler vectorizer still remains effective. This mode can also be selected by the programmer with the `MIPP_NO_INTRINSICS` macro.

MIPP supports the following data types: `double`, `float`, `int64_t`, `int32_t`, `int16_t` and `int8_t`. It also supplies an aligned memory allocator, to be used with types such as the `std::vector<T,A>` vector container from the C++ standard library (where `T` is the vector element type and `A` the allocator). The alignment requirements are

[1] MIPP source code: https://github.com/aff3ct/MIPP

not guaranteed by the default C++ memory allocator. The MIPP memory allocator can be used as follows:

```
std::vector<T,mipp::allocator> aligned_data;
```

and shortened like this: `mipp::vector<T>`.

MIPP comes with a comprehensive unitary test suite to validate new instruction set ports and new feature implementations. It has successfully been tested with the following minimum compiler versions: `g++-4.8`, `clang++-3.6`, `icpc15` and `msvc14.0`.

MIPP implements a generic reduction operator based on a reduction tree, which would be tedious to write by the application programmer, due to the sequence of heterogeneous shuffle instructions it implies. The computational complexity of this algorithm is $O(\log_2(N))$, with N the number of elements in a register. It can operate on `mipp::reg`, `mipp::Reg<T>` and `std::vector<T>`. It can also work on dynamically allocated arrays, provided the length of the array is a multiple of the vector register size. Since the function passed to the reduction operator is resolved at the compile time, the code remains efficient. Any function with the following prototype can be used as the reduction function:

```
mipp::Reg<T> func(mipp::Reg<T>, mipp::Reg<T>)
```

E.g., the code below computes the smallest element in a register:

```
mipp::Reg<float> r = {4.f, 2.f, 1.f, 3.f};
float min = mipp::Reduction<mipp::min>::sapply(r);
```

The `min` scalar variable will be assigned `1.f` as the result. For convenience, a set of functions is predefined, based on this generic reduction feature: `hadd`, `hsub`, `hmul` and `hdiv`.

3 ERROR CORRECTION CODING IN 5G

It is now viable to implement channel coding algorithms on programmable processors under real time constraints. This kind of optimized software implementation could be used in some scenarios of the future 5G mobile communication standards. Beside the real time implementation, it is also necessary to predict the error correction capability of a channel code. To this end, the classical method is to use Monte Carlo simulations in which noise samples are added to the coded data in such a way that one can estimate the residual bit error rate after the channel decoding process. The generation of uncorrelated random data is a computationally intensive task and also requires optimization. In this section, we present several algorithms that will be used in future 5G communication systems together with a random sample generation algorithm and a quantizer used for simulation of communication systems. For each algorithm, a description of the MIPP-optimized implementation is provided. Some speedup measurements are given for different instruction sets. All the source codes are implemented in AFF3CT[2], an Open-source library dedicated to the channel coding.

3.1 Experimentation Protocol

Four architectures are considered for performance results, summarized in Table 1. The Cortex A15 is used to evaluate the NEON instruction set in 32-bit. The Cortex A72 is used to evaluate the 64-bit NEON instructions for Figure 3. The Core i5 is used for both SSE and AVX benchmarks. The Xeon Phi is used for AVX-512 instructions. Source codes are compiled with the GNU C++ 5 compiler using the common flags: `-O3 -funroll-loops`. The additional architecture specific flags are: 1) `-march=armv7-a -mfpu=neon-vfpv4` on Cortex A15, 2) `-march=armv8-a` on Cortex A72, 3) `-msse4.2` for SSE or `-mavx2 -mfma` for AVX on Core i5, 4) `-mavx512f -mfma` on Xeon Phi. All experiments have been performed in single-threaded. All studied problem sizes fit into the last level cache (LLC) of CPUs. The references for the speedup computations are always sequential versions of the SIMD codes. Those reference versions can be auto-vectorized by the compiler, thus a reference version is compiled for each SIMD instruction set.

[2]AFF3CT source code: https://github.com/aff3ct/aff3ct

Table 1: Specifications of the target processors.

Name	Exynos5422	RK3399	Core i5-5300U	Xeon Phi 7230
Year	2014	2016	2015	2016
Vendor	Samsung	Rockchip	Intel	Intel
Arch.	ARMv7 Cortex A15	ARMv8 Cortex A72	Broadwell	Knights Landing
Cores/Freq.	4/2.0 GHz	2/1.6 GHz	2/2.3 GHz	64/1.3 GHz
LLC	2 MB L2	1 MB L2	3 MB L3	32MB L2
TDP	~4 W	~2 W	15 W	215 W

3.2 Box-Muller Transform

Monte Carlo simulations of digital communication systems provide an empirical way to evaluate error correction performance of the system. In this kind of simulations, the channel is modeled as a white Gaussian noise added to the encoded data. This noise generation can be split in two parts: 1) the uniformly-distributed random variable generation, 2) the transformation to a Gaussian random variable. An uniform noise can be generated by a pseudo random number generator (PRNG) like the Mersenne Twister 19937 (MT19937). Then, the Box-Muller method [2] transforms uniformly distributed random numbers into normally distributed random numbers.

Suppose U_1 and U_2 are independent random variables uniformly distributed in $]0, 1]$:

$$z_1 = \sqrt{-2\log U_1}.\cos(2\pi.U_2)),\ z_2 = \sqrt{-2\log U_1}.\sin(2\pi.U_2)).$$

Then, z_1 and z_2 are independent and normally distributed samples.

Listing 2: Box-Muller Transform MIPP kernel.

```
1 void BoxMullerTransform(const std::vector<float> &uniRand
2                          std::vector<float> &norRand) {
3   constexpr auto N = mipp::N<float>();
4   const auto nElmts = uniRand.size();
5   const auto twoPi = 2.f * 3.141592f;
6   for (auto i = 0; i < nElmts; i += N * 2) {
7     const auto u1 = mipp::Reg<float>(&uniRand[   i]);
8     const auto u2 = mipp::Reg<float>(&uniRand[N +i]);
9     const auto radius = mipp::sqrt(mipp::log(u1) * -2.f);
10    const auto theta  = u2 * twoPi;
11    mipp::Reg<float> sintheta, costheta;
12    mipp::sincos(theta, sintheta, costheta);
13    auto z1 = radius * costheta;
14    auto z2 = radius * sintheta;
15    z1.store(&norRand[   i]);
16    z2.store(&norRand[N +i]);
17 } }
```

Listing 2 presents a MIPP implementation of the Box-Muller transform. `uniRand` is a vector of independent and uniformly distributed random numbers (for instance generated with the MT19937 PRNG).

norRand is a vector of independent and normally distributed random numbers. The code stresses SIMD units with multiplications, mipp::sqrt and mipp::sincos calls.

Table 2: AWGN channel with MIPP.

	NEON	SSE	AVX	AVX-512
SIMD size	4	4	8	16
T/P (Mb/s)	40.9	107.4	178.3	95.1
Speedup	×3.1	×2.3	×4.2	×14.4

The MIPP wrapper helps to write a readable code without sacrificing the performance. It also gives the opportunity to compile this same source code for various architectures. Table 2 presents the measured speedups with the same MIPP code compiled for NEON, SSE, AVX and AVX-512, compared to the sequential code (can be auto-vectorized). It also shows the actual throughput (T/P) for each instruction set. The conversion of floating-point format from single precision to double precision only requires to replace the float keyword by double. The ability to switch seamlessly from one data type to another is clearly a strength of the MIPP library. In the source code of the AFF3CT library, the register type is based on a generic type name T (mipp::Reg<T>). This way the same source code can work on float or on double data types.

3.3 Quantizer

During the implementation of ECC decoders, a common step is to convert the floating-point representation into a fixed-point representation. This is necessary after the reception of the noisy channel information representing *Logarithmic Likelihood Ratios* (LLRs) and encoded as real values. The reduction of the LLRs precision (from 32 bits floating-point to 16 or 8 bits fixed-point) does not significantly affect error correction performance and provides more SIMD parallelism. The quantizer computes:

$$y_{s,v} = \min(\max(2^v.y \pm 0.5, -2^{s-1} + 1), 2^{s-1} - 1),$$

with y the current floating-point value, s the number of bits of the quantized number, including v bits for the fractional part.

Table 3: Quantizer speedups with MIPP.

	NEON	SSE	AVX
SIMD size	4-16	4-16	8-32
T/P (Mb/s)	300.6	3541.4	5628.3
Speedup	×4.6	×15.6	×25.8

The associate sequential code is presented in Listing 3. The code converts float (32-bit floating-point number) to int8_t (8-bit signed integer). Although the scalar code is fairlty simple, the compiler fails to auto-vectorize the for-loop. MIPP allows to convert floating-point data types to integers with the mipp::cvt function. It also compresses larger data types into shorter ones with the mipp::pack function. The MIPP code is presented in Listing 4. It performs explicit data types packaging, while in the sequential code, this operation is done implicitly by the (int8_t) cast. Table 3 presents the obtained speedups with MIPP. For this specific case study the speedups are significant for SSE and AVX. They are less important with the NEON instruction set but still non-negligible. We do not provide results for AVX-512, since an AVX-512BW compatible CPU would be required and the Xeon Phi is not.

Listing 3: Sequential implementation of the quantizer.

```
void scalarQuantizer(const std::vector<float > &Y1,
                           std::vector<int8_t> &Y2,
                     const unsigned s, const unsigned v) {
  const auto  K = Y1.size();
  const float factor = 1 << v;
  const float qMax = (1 << (s-2)) + (1 << (s-2)) -1;
  const float qMin = -qMax;
  for (auto k = 0; k < K; k++) {
    // q = 2^v * y +- 0.5
    float q = std::round(factor * Y1[k]);
    // saturation
    Y2[k] = (int8_t)std::min(std::max(q, qMin), qMax);
} }
```

Listing 4: SIMD implementation of the quantizer.

```
void SIMDQuantizer(const std::vector<float > &Y1,
                         std::vector<int8_t> &Y2,
                   const unsigned s, const unsigned v) {
  constexpr auto N = mipp::nElReg<float>();
  const auto K = Y1.size();
  const auto factor = mipp::Reg<float>(1 << v);
  const float qMax = (1 << (s-2)) + (1 << (s-2)) -1;
  const float qMin = -qMax;
  for (auto k = 0; k < K; k += 4 * N) {
    // implicit loads and q = 2^v * y +- 0.5
    auto q32_0 = mipp::round(factor * &Y1[k + 0*N]);
    auto q32_1 = mipp::round(factor * &Y1[k + 1*N]);
    auto q32_2 = mipp::round(factor * &Y1[k + 2*N]);
    auto q32_3 = mipp::round(factor * &Y1[k + 3*N]);
    // convert float to int32_t
    auto q32i_0 = mipp::cvt<int32_t>(q32_0);
    auto q32i_1 = mipp::cvt<int32_t>(q32_1);
    auto q32i_2 = mipp::cvt<int32_t>(q32_2);
    auto q32i_3 = mipp::cvt<int32_t>(q32_3);
    // pack four int32_t in two int16_t
    auto q16i_0 = mipp::pack<int32_t,int16_t>(q32i_0, q32i_1);
    auto q16i_1 = mipp::pack<int32_t,int16_t>(q32i_2, q32i_3);
    // pack two int16_t in one int8_t
    auto q8i = mipp::pack<int16_t,int8_t>(q16i_0, q16i_1);
    // saturation
    auto q8is = mipp::sat(q8i, qMin, qMax);
    q8is.store(&Y2[k]);
} }
```

3.4 LDPC Codes Decoding

LDPC codes is a family of channel codes that is well spread in current digital communication systems. They have been chosen in many communication standards (Wifi, WiMAX, DVB-S2, 10Gbps Ethernet, etc.). They were also selected for the future 5G standard data transport.

In this section the Min-Sum decoder for LDPC codes is presented. As shown in Figure 1, an LDPC code can be represented in the form of a Tanner graph. The circles, denoted as variable nodes, represent the LLRs (the noisy estimation of the bits in the received frames). The squares, denoted as parity check nodes, represent the parity constraints that the variable nodes have to verify. For instance, the check node a (CN_a) is connected to the variable nodes 1, 4, 5, 7 and

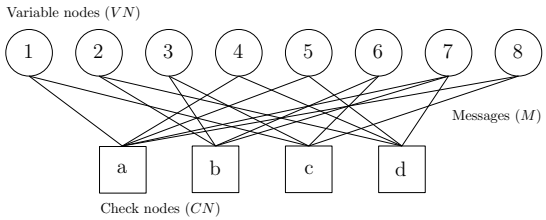

Figure 1: Tanner graph of a simple parity check H matrix

8 ($VN_1, VN_4, VN_5, VN_7, VN_8$). It means that the corresponding bits U_1, U_4, U_5, U_7, U_8 have to respect a parity constraint: $U_1 \oplus U_4 \oplus U_5 \oplus U_7 \oplus U_8 = 0$. A codeword is valid only if it respects all the parity constraints defined by the check nodes. The LDPC code can be also represented by a *parity check matrix*:

$$H = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

The Min-Sum decoder is an iterative message passing algorithm based on the Tanner graph representation. Probabilistic messages (M) are exchanged between the variable nodes and check nodes iteratively. Variable nodes and check nodes apply an *update rule* to compute the outgoing messages from the incoming messages. In this section, the Min-Sum update rule is considered as well as an horizontal layered scheduling. The original version of the Min-Sum algorithm works on floating-point values, but it has been shown that fixed-point simplifications have very similar decoding performance. Moreover, a fixed-point representation enables to pack more elements into SIMD registers.

Table 4: LDPC decoder speedups with `MIPP`.

	NEON	SSE	AVX
SIMD size	8	8	16
T/P (Mb/s)	8.3	30.3	53.2
Speedup	×9.7	×8.8	×15.2

Listing 5 shows a 16-bit fixed-point LDPC decoder. This decoder works on several frames at once. Each element of the SIMD registers corresponds to an element of a specific frame. This approach is called the *inter-frame* vectorization. This strategy maximizes decoder throughput at the expense of latency. Notice that the data type can be switched from `int16_t` to `int8_t`, `int32_t`, `float` or `double`. This `MIPP` feature is important for digital communication: adapting the data type without changing the source code enables to address varying constraints with a single source code. Table 4 presents speedups obtained with `MIPP`. Ten iterations are performed and a stop criterion was implemented for the tests based on parity check constraints (not shown in Listing 5). The H matrix comes from the IEEE 802.3an standard (10Gbps Ethernet). Speedups are close to the SIMD width. In `NEON` and `SSE` they even exceed it. Such result can be explained by an optimized memory management compared to the sequential version of the code.

Listing 5: LDPC decoder implementation with `MIPP`.

```
1 void DecBP(const std::vector<std::vector<int>>   &H
2                 std::vector<mipp::Reg<int16_t>> &VN
3                 std::vector<mipp::Reg<int16_t>> &M,
4                 std::vector<mipp::Reg<int16_t>> &C
5            const unsigned                       nIte) {
6   constexpr auto N = mipp::nElReg<int16_t>();
7   const auto max = std::numeric_limits<int16_t>::max();
8   const auto zeroMsk = mipp::Msk<N>(false);
9   const auto zero = mipp::Reg<int16_t>(0);
10  for (auto i = 0; i < nIte; i++) {
11    auto mRead = 0, mWrite = 0;
12    for (auto c = 0; c < H.size(); c++) {
13      auto sign = zeroMsk;
14      auto min1 = mipp::Reg<int16_t>(max);
15      auto min2 = mipp::Reg<int16_t>(max);
16      for (auto v = 0; v < H[c].size(); v++) {
17        C[v] = VN[H[c][v]] - M[mRead++];
18        auto cabs = mipp::abs(C[v]);
19        auto ctmp = min1;
20        sign ^= mipp::sign(C[v]);
21        min1  = mipp::min(min1,           cabs       );
22        min2  = mipp::min(min2, mipp::max(cabs, ctmp));
23      }
24      auto cst1 = mipp::blend(zero, min2, zero > min2);
25      auto cst2 = mipp::blend(zero, min1, zero > min1);
26      for (auto v = 0; v < H[c].size(); v++) {
27        auto cval = C[v];
28        auto cabs = mipp::abs(cval);
29        auto cres = mipp::blend(cst1, cst2, cabs == min1);
30        auto csig = sign ^ mipp::sign(cval);
31             cres = mipp::copysign(cres, csig);
32        M[mWrite++] = cres;
33        VN[H[c][v]] = C[v] + cres;
34  } } } }
```

3.5 Polar Codes Decoding

Polar codes have been introduced by Arıkan in 2008 [1]. They have also been selected as channel codes in the future 5G standard to improve the reliability of the data control channels. We present the Successive Cancellation (SC) polar decoding algorithm. As Min-Sum for LDPC codes, SC decoding is also a message passing algorithm. Here, messages propagate on a binary tree in a depth-first scheduling. Figure 2 shows a polar code tree representation. Descending messages are LLRs, ascending messages are bits. Each node applies

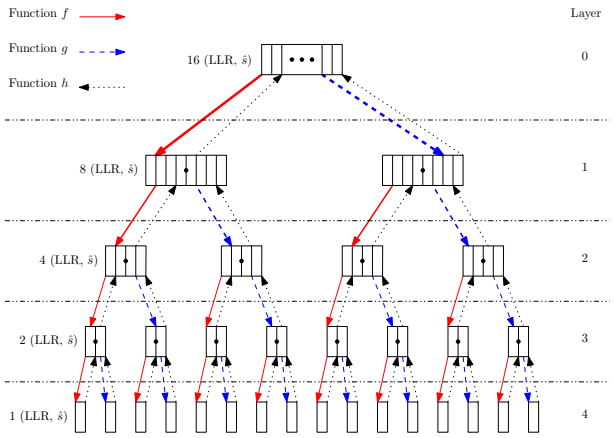

Figure 2: Tree representation of the SC polar decoding

Listing 6: MIPP implementations of f, g and h functions.

```
mipp::Reg<int8_t> f_simd(const mipp::Reg<int8_t> &la,
                         const mipp::Reg<int8_t> &lb) {
  auto abs_min = mipp::min(mipp::abs(la), mipp::abs(lb));
  auto sign    = mipp::sign(la ^ lb);
  return mipp::neg(abs_min, sign);
}
mipp::Reg<int8_t> g_simd(const mipp::Reg<int8_t> &la,
                         const mipp::Reg<int8_t> &lb,
                         const mipp::Reg<int8_t> &sa) {
  return mipp::neg(la, sa) + lb;
}
mipp::Reg<int8_t> h_simd(const mipp::Reg<int8_t>& sa,
                         const mipp::Reg<int8_t>& sb) {
  return sa ^ sb;
}
```

an update rule on the messages. When going down to the left, rule f is applied; when going down to the right, rule g is applied, when going up, rule h function is applied:

$$\begin{cases} f(\lambda_a, \lambda_b) & = & sign(\lambda_a.\lambda_b).\min(|\lambda_a|, |\lambda_b|) \\ g(\lambda_a, \lambda_b, \hat{s}_a) & = & (1 - 2\hat{s}_a)\lambda_a + \lambda_b \\ h(\hat{s}_a, \hat{s}_b) & = & (\hat{s}_a \oplus \hat{s}_b, \hat{s}_b) \end{cases}$$

The number of elements to compute per node is halved from one layer to the next. E.g. in Layer 1, 8 independent elements can be computed in the f, g, h functions while in the Layer 2, there are only 4 independent elements to compute by node. Listing 6 presents an 8-bit fixed-point SIMD implementation of rules f, g, h. They compute multiple elements in the same frame to exploit the *intra-frame* parallelism. This strategy has a lower throughput than inter-frame strategy but it also a lower latency. It is constrained by the decoding algorithm, which has a limited intrinsic parallelism. Indeed, at some point in the lowest layers of the tree, there is not enough parallelism anymore, and sequential f, g, h functions have to be used instead. As for the LDPC decoder, the data type of the SC decoder can be switched from `int8_t` to `int16_t`, `int32_t`, `float` or `double`.

Table 5: SC decoder speedups with MIPP.

	NEON	SSE	AVX
SIMD size	16	16	32
T/P (Mb/s)	148.7	528.3	483.0
Speedup	×3.1	×4.4	×3.8

In typical SC applications, some tree cuts can be applied statically out of the decoder selected parameters. Such tree cut instances significantly reduce the number of traversed nodes in the low levels of the tree, where parallelism is low. Such a tree cut version (Fast-SSC [12]) has been used for the experiments presented here. Table 5 shows the obtained speedups. This time, even if the potential parallelism is high (16 for NEON and SSE, 32 for AVX), the measured speedups does not exceed 4.4, due to the remaining low parallelism nodes. Also the sequential implementation of the code has been almost fully auto-vectorized. The SSE code performs better than the AVX one, because of the nature of the algorithm. Actually, using larger SIMD registers is good for nodes with enough parallelism, but this reduces the total number of nodes which can be effectively vectorized. In these sub-optimal cases, the SIMD loads and stores cannot be aligned anymore and sometime a sequential code replaces the SIMD implementation. As a consequence, for the selected codewords, it is more advantageous to use SSE than AVX.

4 RELATED WORKS

Many SIMD programming solutions have been surveyed in [24] to take advantage of modern instruction sets. The existing alternatives can be decomposed into three main models: 1) intrinsics or assembly code; 2) dedicated language; and 3) dedicated library. The intrinsics or assembly approaches are non-portable, low-level solutions which target specific architectures. They offer maximum control to take advantage of instruction set specificities, and to fine tune register usage. However, it is quite difficult to develop and maintain a low-level code in the long run. Some languages have been designed to provide programmers with SIMD programming constructs. Many of them are based on general purpose languages extended with some kinds of annotation mechanism (e.g. pragmas) such as OpenMP [21], Cilk Plus [25] or ispc [23]. They offer higher expressiveness, better portability and generally more readable code, at the expense of less programmer control, and vectorization performance. More specialized languages, such as OpenCL [13], enable the programmer to retain more control, as the counterpart of writing some more specific code. In this paper, the focus is given to the library approach since we want to maximize performance, maximize portability and deal with existing C++ codes. In order to let the compiler inline library calls, which is critical for the intended SIMD programming model purpose, such library are usually header-only. Thus, we refer to them as *wrappers* instead of *libraries*.

4.1 C++ SIMD Wrappers

Table 6 compares various SIMD wrappers. It aims to present an overview of some prominent solutions, though it is by no means exhaustive due to the richness of the SIMD wrapper landscape. Some of the wrappers presented, such as MIPP, Vc, Boost.SIMD, VCL and T-SIMD, have been designed in an academic research context. Some others, simdpp and xsimd, appear to be standalone development efforts by individual programmers or maintainers. Proprietary, closed-source solutions also exist on the market, such as bSIMD, which is an extended version of Boost.SIMD, or the commercial version of VCL. The *Instruction Set* column is broken up into five families among the most widely available on the market: NEON, SSE, AVX, AVX-512 and AltiVec. For the sake of conciseness, we choose not to list all the instruction sets "sub-variants" (such as SSE2, SSE3, etc). simdpp et bSIMD propose the most comprehensive instruction set compatibility. At the other end of the range, xsimd and Boost.SIMD only support Intel SIMD instruction sets. The *Data Type* column of the table summarizes the supported vector element types and precisions. In their public version, and at the time of writing, Vc does not support 8-bit integers, xsimd does not support 8-bit and 16-bit integers and T-SIMD does not support 64-bit data types, to the best of our knowledge. The *Features* column highlights some additional characteristics. The *Math Func.* column indicates which wrapper supports additional mathematical sub-routines, not necessarily available as native CPU instructions (exponential, logarithm, trigonometric functions for instance), and required by algorithms

Table 6: Comparison of various SIMD wrappers.

	General Information				Instruction Set					Data Type						Features		
	Name	Ref.	Start Year	License	SSE 128-bit	AVX 256-bit	AVX-512 512-bit	NEON 128-bit	AltiVec 128-bit	Float 64	Float 32	Integer 64	Integer 32	Integer 16	Integer 8	Math Func.	C++ Technique	Test Suite
Library	MIPP	–	2013	MIT	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	Op. overload.	✓
	VCL	[10]	2012	GNU GPL	✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓	Op. overload.	N/A
	simdpp	[14]	2013	Boost Software	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	Expr. templ.	✓
	T-SIMD	[19]	2016	Open-source	✓	✓	✗	✓	✗	✗	✓	✗	✓	✓	✓	✗	Op. overload.	N/A
	Vc	[15]	2012	BSD-3-Clause	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✗	✓	Op. overload.	✓
	xsimd	[18]	2014	BSD-3-Clause	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗	✓	Op. overload.	N/A
	Boost.SIMD	[8]	2012	Boost Software	✓	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	Expr. templ.	✓
	bSIMD	[7]	2017	Non-free	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Expr. templ.	✓

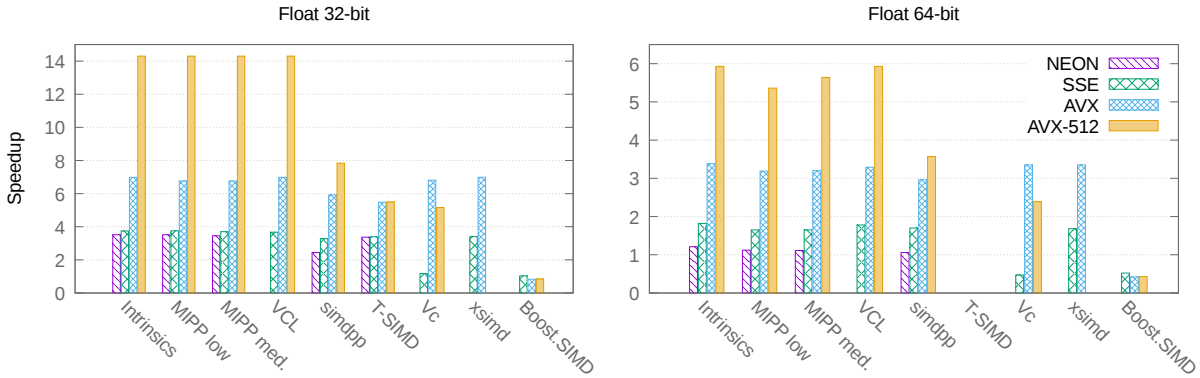

Figure 3: Speedups over the Mandelbrot naive auto-vectorized implementation

such as the Box-Muller Transform (see Section 3.2). The C++ *Technique* column indicates whether the wrapper is designed as an expression template framework, or whether it relies on operator overloading techniques. The expression template feature is a powerful technique to automatically drive the rewriting of whole arithmetic expressions into SIMD hardware instructions or instruction sequences. For instance if the user writes `d = a * b + c`, the wrapper can automatically match a *fused multiply and add* instruction (FMA). `Boost.SIMD` and `bSIMD` extensively use this technique [7, 8]. The drawbacks are that the source code complexity of the wrapper is dramatically increased. `Boost.SIMD` and `bSIMD` have a dependency on the Boost framework to build, and currently available C++ compilers produce huge amounts of arcane error messages at the slightest mistake in the end user program. For these reasons, we decided not to base `MIPP` on the expression template technique. As mentioned in Section 2.3, maintaining SIMD wrappers, and porting them to new instruction sets is error prone by nature, due to the large number of routines, cryptic intrinsics names, and specific instruction set details. A comprehensive testing suite is therefore critical to validate new development, optimizations and ports on new instruction sets. This is why `MIPP`, as well as `Vc`, `Boost.SIMD`, `simdpp` and `bSIMD` come with their own test suites. We have not found similar test suites in the software distributions of `VCL`, `xsimd` and `T-SIMD`; however, test suites might be in use internally, within the development teams of these wrappers.

4.2 Qualitative and Quantitative Comparisons

We now compare `MIPP` with the open-source wrappers presented above, both qualitatively for our error correction code purpose, and quantitatively on a well known benchmark the computation of the Mandelbrot set, to prevent as much as possible the risk of unfairness of the port on each wrapper. This problem is compute-bound. The chosen implementation relies on a floating-point representation (available online[3]). Figure 3 presents the speedups obtained on various instruction sets. `SSE` stands for `SSE4.2`, `NEON` stands for `NEONv2` (includes the FMA instructions), `AVX` stands for `AVX2+FMA3` and `AVX-512` stands for `AVX-512F` (with FMA instructions). The `FMA` benefit ranges from 17% (`AVX2`) to 26% (`AVX-512`). An SIMD with intrinsics version has been hand-coded for each specific instruction set. The intrinsics version is considered the "*golden*" model.

`Boost.SIMD` only supports the `SSE` instruction set, even when the code is compiled with one of the `AVX` or `AVX-512` flags. It is insufficient for our channel coding processing purpose. The `Boost.SIMD` wrapper performance results that were obtained are disappointing. The sequential Mandelbrot kernel does an early exit in the innermost loop, as soon as the divergence of the sequence is detected for the input coordinates. We were unable to SIMDize this early termination with `Boost.SIMD`, because the `boost::simd::any` function was not available in the GitHub repository at the time of writing. **`xsimd`** achieves performance close to the intrinsic version in `SSE` and `AVX`. However, it currently lacks `NEON` and `AVX-512` support. Moreover, it does not support small 8-bit and 16-bit integers, needed

[3] Mandelbrot set source code: https://gitlab.inria.fr/acassagn/mandelbrot

for Successive Cancellation decoders (see Section 3.5). **Vc** is one of the earliest developed SIMD C++ wrapper. We used Branch 1.3 for the performance measurements, the latest stable branch at this time. Vc includes a lot of of features compared to the other wrappers; but it lacks support for NEON and AVX-512 (which are currently being developed). Performance results are on par with the best contenders for AVX. However, a slowdown is observed for SSE. Note: For AVX-512, since the support is not yet available in the stable version, we used the capability of Vc to generate AVX2 code in order to produce the sample points for AVX-512 series. The results are likely to improve once the full AVX-512 support is release in a subsequent stable version. **T-SIMD** is a wrapper primarily designed for image processing purpose. It performs well in 32-bit NEON, SSE and AVX but it lacks from AVX-512. Support of the 64-bit types is not planed since it is not useful in traditional image computations. **simdpp** supports an impressive number of instruction sets. This may explain why it does not support mathematical functions so far. It matches the performance of the other wrappers for NEON and SSE, but falls behind for AVX, and even more for AVX-512. **VCL** is a high performance wrapper and perhaps the most feature rich for x86 SIMD at this time. It gives a lot of control to the developer and it is well documented. The obtained performance are on the same level as hand-written intrinsics. However, it is not yet available on NEON. For MIPP we have tested both the lower-level programming interface and the medium-level programming interface of our MIPP wrapper, mainly to detect potential overheads when using the medium level interface instead of the lower one. The obtained results do not show any performance penalties when using MIPP medium level interface. The obtained speedups are close to the intrinsics version.

MIPP corresponds to a programming model close to the intrinsics, with some adaptation to architectures. Still, a high performance code requires that the developer knows how to decompose efficiently some computation with the SIMD instructions. Between AVX-512 and SSE or NEON for instance, several implementations of the same code are possible. MIPP offers to the programmer the control on the intrinsics taken and ensures portability.

5 CONCLUSION

This paper introduces MIPP, a C++ wrapper for SIMD intrinsics. It is designed to fit recent digital communication algorithms, especially channel decoding algorithms such as LDPC or Polar code methods. It offers high portability: the same code can be compiled on different target architectures and still offer high performance, adapted for both simulation of channel decoding on high-end machines and embedded software decoding. The decoding accuracy is set simply by choosing the vector element types used as template parameters. Despite its application-oriented design, MIPP shows state-of-the-art performance on the Mandelbrot generation algorithm.

As future works, we plan to wrap the new ARM *Scalable Vector Extension* (SVE) [29], a vector-length agnostic architecture, thus the main challenge will be to deal with unknown vector sizes at compile time. However, the philosophy of SVE somewhat coincides with MIPP's: a same code can accommodate and mix various vector sizes. Moreover, MIPP's masks would likely map on SVE's predicate registers straight away.

ACKNOWLEDGMENTS

The authors would like to thank Mathieu Léonardon for providing advices and a comprehensive review on this paper. Experiments presented in this paper were carried out using the PLAFRIM experimental testbed.

REFERENCES

[1] E. Arikan. 2009. Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels. *IEEE TIT* (2009). https://doi.org/10.1109/TIT.2009.2021379
[2] G. E. P. Box, M. E. Muller, et al. 1958. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics* (1958).
[3] A. Cassagne et al. 2016. Beyond Gbps Turbo decoder on multi-core CPUs. In *ISTC*. IEEE. https://doi.org/10.1109/ISTC.2016.7593092
[4] A. Cassagne, O. Aumage, C. Leroux, D. Barthou, and B. Le Gal. 2016. Energy Consumption Analysis of Software Polar Decoders on Low Power Processors. In *EUSIPCO*. IEEE. https://doi.org/10.1109/EUSIPCO.2016.7760327
[5] A. Cassagne, B. Le Gal, C. Leroux, O. Aumage, and D. Barthou. 2016. An Efficient, Portable and Generic Library for Successive Cancellation Decoding of Polar Codes. In *LCPC* (2016-01-01). Springer. https://doi.org/10.1007/978-3-319-29778-1_19
[6] Ericsson. 2015. *Cloud RAN - The Benefits of Cirtualization, Centralisation and Coordination.* Technical Report. Ericsson. https://www.ericsson.com/assets/local/publications/white-papers/wp-cloud-ran.pdf
[7] P. Esterie et al. 2012. Exploiting Multimedia Extensions in C++: A Portable Approach. *IEEE CS&E* (2012). https://doi.org/10.1109/MCSE.2012.96
[8] P. Estérie, M. Gaunard, J. Falcou, J. T. Lapresté, and B. Rozoy. 2012. Boost.SIMD: Generic programming for portable SIMDization. In *PACT*. IEEE.
[9] ETSI. 2017. 3GPP-TS38.212 - Multiplexing and chan. coding (R. 15). (2017).
[10] A. Fog. 2017. Vector Class Lib. (2017). www.agner.org/optimize/#vectorclass
[11] P. Giard, G. Sarkis, C. Leroux, C. Thibeault, and W. J. Gross. 2016. Low-Latency Software Polar Decoders. *JSPS* (2016). https://doi.org/10.1007/s11265-016-1157-y
[12] P. Giard, G. Sarkis, C. Thibeault, and W. J. Gross. 2014. Fast Software Polar Decoders. In *ICASSP*. IEEE. https://doi.org/10.1109/ICASSP.2014.6855069
[13] L. Howes. 2015. The OpenCL Specification. (2015). https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf Version 2.1, Revision 23.
[14] P. Kanapickas. 2017. libsimdpp. (2017). https://github.com/p12tic/libsimdpp
[15] M. Kretz and V. Lindenstruth. 2012. Vc: A C++ Library for Explicit Vectorization. *Software: Practice and Experience* (2012). https://doi.org/10.1002/spe.1149
[16] B. Le Gal et al. 2015. High-Throughput LDPC Decoder on Low-Power Embedded Processors. *IEEE COMML* (2015). https://doi.org/10.1109/LCOMM.2015.2477081
[17] B. Le Gal and C. Jego. 2016. High-Throughput Multi-Core LDPC Decoders Based on x86 Processor. *IEEE TPDS* (2016). https://doi.org/10.1109/TPDS.2015.2435787
[18] J. Mabille. 2017. xsimd. (2017). https://github.com/JohanMabille/xsimd
[19] R. Möller. 2016. *Design of a Low-Level C++ Template SIMD Library.* Technical Report. Bielefeld University, Faculty of Technology, Computer Engineering Group. http://www.ti.uni-bielefeld.de/html/people/moeller/tsimd_warpingsimd.html
[20] N. Nikaein. 2015. Processing RAN Functions in the Cloud: Critical Issues and Modeling. In *MCS*. ACM. https://doi.org/10.1145/2802130.2802136
[21] OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface. (2013). http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
[22] H. Paul, D. Wübben, and P. Rost. 2015. Implementation and Analysis of Forward Error Correction Decoding for Cloud-RAN Systems. In *ICCW*. IEEE. https://doi.org/10.1109/ICCW.2015.7247588
[23] M. Pharr and W. R. Mark. 2012. ispc: A SPMD Compiler for High-Performance CPU Programming. In *InPar*. IEEE. https://doi.org/10.1109/InPar.2012.6339601
[24] A. Pohl, B. Cosenza, M. A. Mesa, C. C. Chi, and B. Juurlink. 2016. An Evaluation of Current SIMD Programming Models for C++. In *WPMVP*. ACM, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2870650.2870653
[25] A. D. Robison. 2013. Composable Parallel Patterns with Intel Cilk Plus. *IEEE CS&E* (2013). https://doi.org/10.1109/MCSE.2013.21
[26] V. Q. Rodriguez and F. Guillemin. 2017. Towards the Deployment of a Fully Centralized Cloud-RAN Architecture. In *IWCMC*. IEEE. https://doi.org/10.1109/IWCMC.2017.7986431
[27] P. Rost et al. 2014. Cloud Technologies for Flexible 5G Radio Access Networks. *IEEE Comm. Magazine* (2014). https://doi.org/10.1109/MCOM.2014.6898939
[28] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross. 2016. Fast List Decoders for Polar Codes. *IEEE JSAC* (2016). https://doi.org/10.1109/JSAC.2015.2504299
[29] N. Stephens et al. 2017. The ARM Scalable Vector Extension. *IEEE Micro* (2017). https://doi.org/10.1109/MM.2017.35
[30] M. Wu et al. 2013. HSPA+/LTE-A Turbo Decoder on GPU and Multicore CPU. In *ACSSC*. IEEE. https://doi.org/10.1109/ACSSC.2013.6810402
[31] Suiping Zhang, Rongrong Qian, Tao Peng, Ran Duan, and Kuilin Chen. 2012. High Throughput Turbo Decoder Design for GPP Platform. In *CHINACOM*. IEEE. https://doi.org/10.1109/ChinaCom.2012.6417597