SIMDization with MIPP

My Intrinsics ++

Denis Barthou
Inria Storm team, Bordeaux

STORM
Static Optimizations – Runtime Methods

Inria

MIPP: a Portable C++ SIMD Wrapper

SIMD C++ wrapper

- Maximizes code portability: Intel SSE, AVX, AVX2, AVX512, Neon, SVE (ongoing) and Risc-V (ongoing)
- Improves expressiveness over intrinsics
- Programming model **close to machine** Whenever possible, **one statement = one intrinsic**
- One variable = one register allocation

Open-source: https://github.com/aff3ct/MIPP

- Typed registers, polymorphic operations
- Typing through **object encapsulation** and **operator overloading**

Addition of two vectors in MIPP

```
1  template <typename T>
2  void vecAdd(const std::vector<T> &A,
3              const std::vector<T> &B,
4                    std::vector<T> &C)
5  {
6    // N elements per SIMD register
7    constexpr int N = mipp::N<T>();
8
9    // sizes verifications
10   assert(A.size() == B.size());
11   assert(A.size() == C.size());
12   assert((A.size() % N) == 0);
13
14   for(auto i = 0; i < A.size(); i += N)
15   {
16     mipp::Reg<T> rA = &A[i]; // SIMD load
17     mipp::Reg<T> rB = &B[i]; // SIMD load
18
19     mipp::Reg<T> rC = rA + rB; // SIMD addition
20
21     rC.store(&C[i]); // SIMD store
22   }
23 }
```

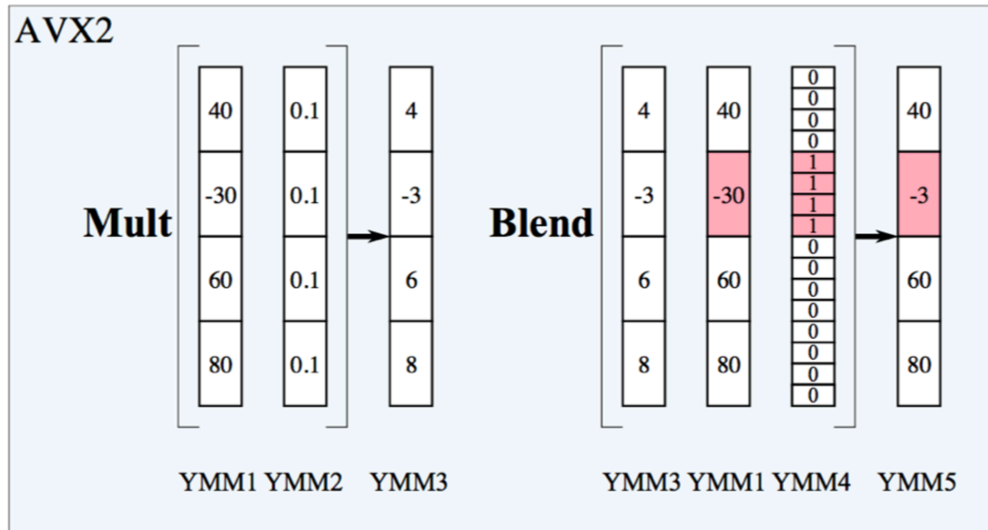
MIPP: a Portable C++ SIMD Wrapper

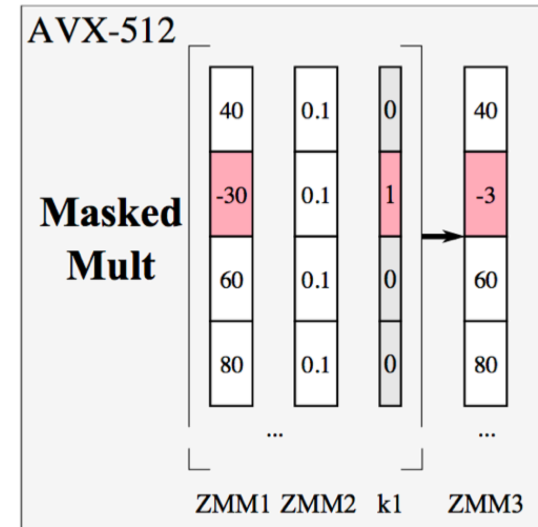
Three main strategies:

- **Operator overloading**
- Expression templates
- DSL and compiler

General Information		Instruction Set				Data Type						Features	
	Name	`SSE` 128-bit	`AVX` 256-bit	`AVX-512` 512-bit	`NEON` 128-bit	**Float** 64	32	**Integer** 64	32	16	8	**Math Func.**	`C++` **Technique**
Library	`MIPP`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Op. overload.
	`VCL`	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	Op. overload.
	`simdpp`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	Expr. templ.
	`T-SIMD`	✓	✓	✗	✓	✗	✓	✗	✓	✓	✓	✗	Op. overload.
	`Vc`	✓	✓	✗	✗	✓	✓	✓	✓	✓	✗	✓	Op. overload.
	`xsimd`	✓	✓	✗	✗	✓	✓	✓	✓	✗	✗	✓	Op. overload.
	`Boost.SIMD`	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	Expr. templ.
	`bSIMD`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Expr. templ.

MIPP: Instruction abstraction





Picture from colfaxresearch.com

```
1 mipp::Reg<          float    > ZMM1 = {   40,  -30,    60,    80};
2 mipp::Reg<          float    > ZMM2 = 0.1; // broadcast
3 mipp::Msk<mipp::N<float>()> k1    = {false, true, false, false};
4
5 // ZMM3 = k1 ? ZMM1 * ZMM2 : ZMM1;
6 auto ZMM3 = mipp::mask<float, mipp::mul>(k1, ZMM1, ZMM1, ZMM2);
7
8 std::cout << ZMM3 << std::endl; // [   40,   -3,    60,    80]
```

MIPP: Portable performance

Polar code decoding (ECC)

```
template <typename T>
mipp::Reg<T> f_SIMD(const mipp::Reg<T> &la,
                    const mipp::Reg<T> &lb)
{
  auto abs_la  = mipp::abs(la);
  auto abs_lb  = mipp::abs(lb);
  auto min_abs = mipp::min(abs_la, abs_lb);
  auto sign    = mipp::sign(la ^ lb);
  // neg(Reg, Msk) = Msk ? -Reg : Reg;
  return mipp::neg(min_abs, sign);
}

template <typename T, int N = mipp::N<T>()>
mipp::Reg<T> g_SIMD(const mipp::Reg<T> &la,
                    const mipp::Reg<T> &lb,
                    const mipp::Msk<N> &sa)
{
  return mipp::neg(la, sa) + lb;
}

template <int N>
mipp::Msk<N> h_SIMD(const mipp::Msk<N>& sa,
                    const mipp::Msk<N>& sb)
{
  return sa ^ sb;
}
```

- **Same code for various data types** (**double**, **float**, **int16_t**, **int8_t**)
- Decodes frame per frame (SIMD **intra-frame strategy**)

	NEON	SSE	AVX
SIMD size	16	16	32
Seq. T/P (Mb/s)	48.0	120.1	127.1
MIPP **T/P** (Mb/s)	148.7	528.3	483.0
Speedup	×3.1	×4.4	×3.8

MIPP: Portable performance

Quantizer for numerical com. (SDR)

```
1  void Quantizer(const std::vector<float > &Y,
2                 std::vector<int8_t> &Ysv,
3                 const unsigned s, const unsigned v) {
4    constexpr auto N = mipp::N<float >();
5    const auto pow2 = mipp::Reg<float >(1 << v);
6    const float qMax = (1 << (s-2)) + (1 << (s-2)) -1;
7    const float qMin = -qMax;
8    for (auto y = 0; y < Y.size(); y += 4 * N) {
9      // implicit loads and q = 2^v * y +/- 0.5
10     auto q32_0 = mipp::round(pow2 * &Y[y + 0*N]);
11     auto q32_1 = mipp::round(pow2 * &Y[y + 1*N]);
12     auto q32_2 = mipp::round(pow2 * &Y[y + 2*N]);
13     auto q32_3 = mipp::round(pow2 * &Y[y + 3*N]);
14     // convert float to int32_t
15     auto q32i_0 = mipp::cvt<int32_t>(q32_0);
16     auto q32i_1 = mipp::cvt<int32_t>(q32_1);
17     auto q32i_2 = mipp::cvt<int32_t>(q32_2);
18     auto q32i_3 = mipp::cvt<int32_t>(q32_3);
19     // pack four int32_t in two int16_t
20     auto q16i_0 = mipp::pack<int32_t,int16_t>(q32i_0,
21                                               q32i_1);
22     auto q16i_1 = mipp::pack<int32_t,int16_t>(q32i_2,
23                                               q32i_3);
24     // pack two int16_t in one int8_t
25     auto q8i = mipp::pack<int16_t,int8_t>(q16i_0, q16i_1);
26     // saturation (psi function)
27     auto q8is = mipp::sat(q8i, qMin, qMax);
28     q8is.store(&Ysv[y]);
29   }
30 }
```

- **Converts** 32-bit floating-point numbers into 8-bit integers
- **Auto-vectorization is very bad** when data conversions are involved
- `MIPP` provides `round`, `cvt` and `pack` functions

	`NEON`	`SSE`	`AVX`
SIMD size	4-16	4-16	8-32
Seq. T/P (Mb/s)	65.3	227.0	218.2
`MIPP` **T/P** (Mb/s)	300.6	3541.4	5628.3
Speedup	×4.6	×15.6	×25.8

MIPP: Portable performance

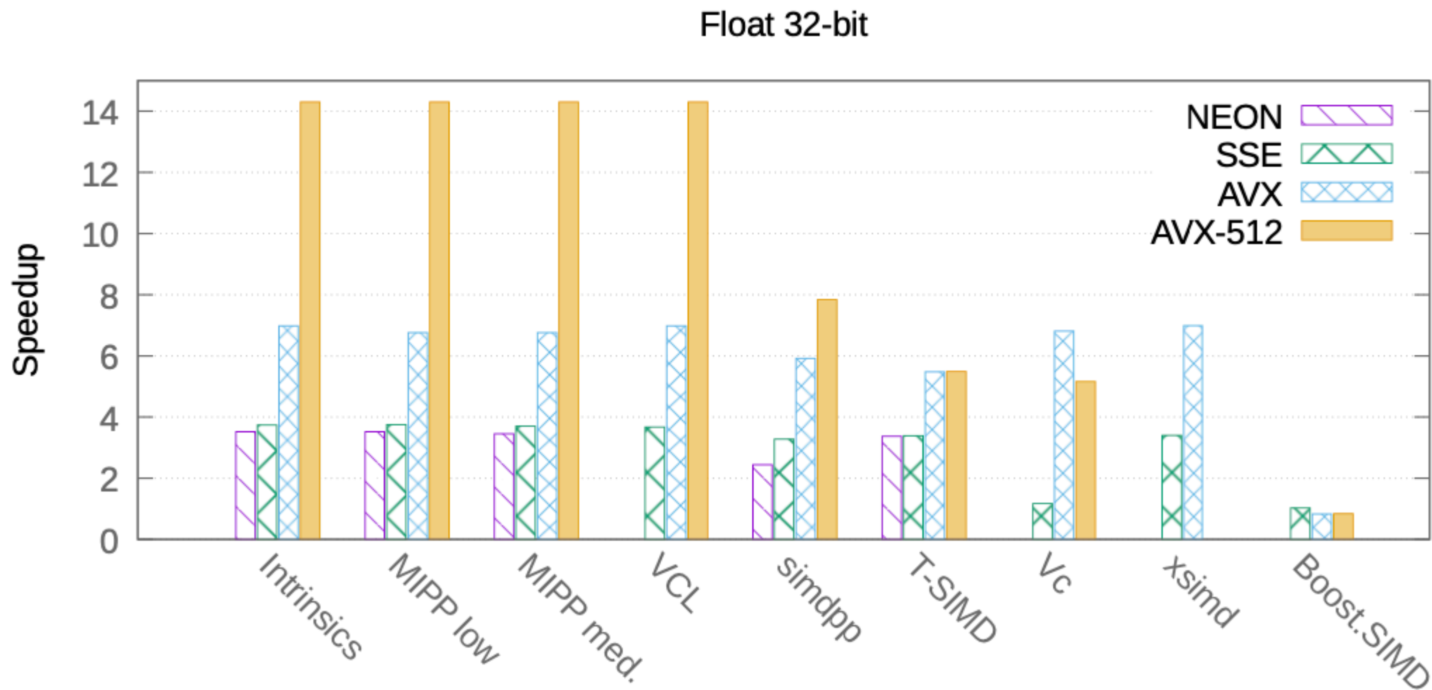
LDPC code decoding (ECC)

```
1  template <typename T>
2  void DecBP(const std::vector<std::vector<unsigned>> &H
3             std::vector<mipp::Reg  <T        >> &VN
4             std::vector<mipp::Reg  <T        >> &M,
5             std::vector<mipp::Reg  <T        >> &C
6             const unsigned                          nIte) {
7    const auto max = std::numeric_limits<T>::max();
8    const auto zero = mipp::Reg<T>(0);
9    for (auto i = 0; i < nIte; i++) {
10     auto mRead = 0, mWrite = 0;
11     for (auto c = 0; c < H.size(); c++) {
12       constexpr auto N = mipp::N<T>();
13       auto sign = mipp::Msk<N>(false);
14       auto min1 = mipp::Reg<T>(max);
15       auto min2 = mipp::Reg<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 } } } }
```

- **Same code for various data types** (`double`, `float`, `int32_t`, `int16_t`)
- Decodes multiple frames (`N`) in parallel (SIMD **inter-frame strategy**)

	NEON	SSE	AVX
SIMD size	8	8	16
Seq. T/P (Mb/s)	0.9	3.4	3.5
MIPP T/P (Mb/s)	8.3	30.3	53.2
Speedup	×9.7	×8.8	×15.2

Comparison with other wrappers



Performance comparison on Mandelbrot, 32bit [WPMVP18]

SIMDization is not enough

- SIMDization is an source instruction level optimization, still need intra-tile optimization
- Old problem, register allocation & scheduling/ unrolling
- OoO should dim impact of these optimizations

```
25      TYPE *B=&vB[0];
26      TYPE *C=&vC[0];
27      TYPE *A=&vA[0];
28      for (int i=0;i<BLOCKI;i+=1) {
29         for (int j=0;j<BLOCKJ;j+=1*nv) {
30            mipp::Reg<TYPE> c00;
31            c00.load(&C[(i+0)*BLOCKJ + j + (0)*nv]);
32            for (int k=0;k<BLOCKK;k+=1) {
33               mipp::Reg<TYPE> a00;
34               a00 = mipp::set1<TYPE>(A[(i+0)*BLOCKK+k+0]);
35               mipp::Reg<TYPE> b00;
36               b00.load(&B[(k+0)*BLOCKJ + j + (0)*nv]);
37               c00 =mipp::fmadd(a00, b00,c00);
38            }
39            c00.store(&C[(i+0)*BLOCKJ + j + (0)*nv]);
40         }
41      }
```

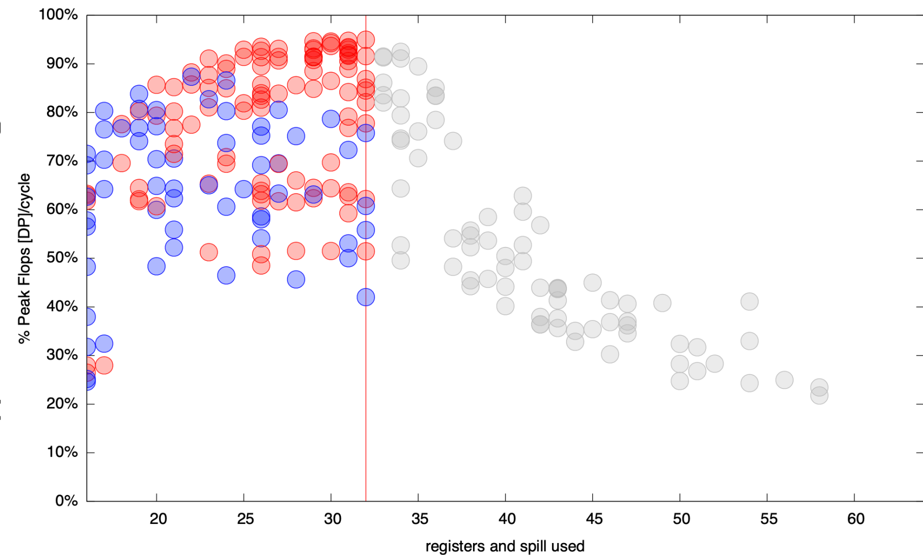
One block of GEMM, with MIPP

SIMDization is not enough

- SIMDization is an source instruction level optimization, still need intra-tile optimization
- Old problem, register allocation & scheduling, unrolling
- OoO should dim impact of these optimizations

Still large factor of performance can be obtained

- Exploration for code generation
- Try unroll



Blue: unrolling should fit in registers. Red: should not but does. Grey: spill
Intel CascadeLake, AVX512, Clang15

Work in progress

Precision evaluation and tradeoff precision/performance

- Change SIMD instruction selection, consider approximate computations
- ARM SVE backend
- Explore codes with different precisions, depending on the computation phases
  - Creates heterogeneity in the computation
  - Changes energy consumption

Questions ?