Single Instruction Multiple Data

ARM NEON and MIPP Wrapper

Sorbonne Université – Master SESI – MU5IN60 – Parallel Programming

Adrien Cassagne

September 25, 2023

Table of Contents

Session Objectives

- Vector and SIMD programming models
  - Origins and definitions
  - Several programming levels
- NEON Instruction Set Architecture (ISA)
  - Intrinsic function calls
- MIPP SIMD Wrapper
  - Memory operations
  - Arithmetic operations
  - Binary operations
  - Logical operations
  - ...

Section 1.1

Vector and SIMD Programming Models

Vector Supercomputers – Part 1

Supercomputers were the first computers to use vector computing to speed up computations. In this type of architecture, instructions encode the number of elements they will process.

The first vector supercomputers:

- The ILLIAC IV machine at the University of Illinois in 1972 ($100 \simeq 150$ MFlop/s)
- The CDC STAR-100 supercomputer in 1974 (36 MFlop/s)
- The Cray-1 supercomputer in 1976 (160 MFlop/s)

Cray-1 (1976)

Vector Supercomputers – Part 2

1 Introduction

Example of scalar instructions:

```
1  loop:
2    load a, [p_a]
3    load b, [p_b]
4    add c, a, b # c[i] = a[i] + b[i]
5    store [p_c], c
6    add p_a, p_a, 4 # 4 bytes, 32-bit
7    add p_b, p_b, 4 # 4 bytes, 32-bit
8    add p_c, p_c, 4 # 4 bytes, 32-bit
9    add i, i, 1
10 jumple i, 10000, loop
```

Example of a vector instruction:

```
1 # c[1..10000] = a[1..10000] + b[1..10000]
2 addv c(1..10000), a(1..10000), b(1..10000)
```

- The `addv` instruction is decoded (and *fetched*) only once!
  - — Gains in pipeline stages *fetch* and *decode*
  - — The instruction is decoded once instead of 10,000 times
  - — Fewer instructions (simpler pointer arithmetic, no loops)
  - — Each element (`c[i]`) is computed sequentially in both cases

Single Instruction Multiple Data (SIMD)

Example of scalar instructions:

```
1  loop:
2    load a, [p_a]
3    load b, [p_b]
4    add c, a, b # c[i] = a[i] + b[i]
5    store [p_c], c
6    add p_a, p_a, 4 # 4 bytes, 32-bit
7    add p_b, p_b, 4 # 4 bytes, 32-bit
8    add p_c, p_c, 4 # 4 bytes, 32-bit
9    add i, i, 1
10 jumple i, 10000, loop
```

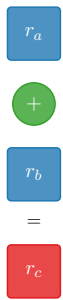
Example of SIMD instructions:

```
1  loop:
2    loadv va, [p_a] # read 4 elements
3    loadv vb, [p_b] # read 4 elements
4    addv vc, va, vb # add 4 elements
5    storev [p_c], vc # write 4 elements
6    add p_a, p_a, 4*4 # 4*4 bytes, 128-bit
7    add p_b, p_b, 4*4 # 4*4 bytes, 128-bit
8    add p_c, p_c, 4*4 # 4*4 bytes, 128-bit
9    add i, i, 4 # increment i of 4
10 jumple i, 10000, loop
```

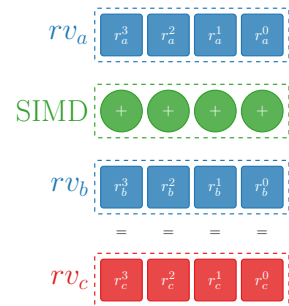
- The `addv`, `loadv` and `storev` instructions perform operations on 4 elements at a time
  - 4 times fewer instructions (potential acceleration of 4)
  - The number of elements in a vector register = the # lanes (here 4)
  - Each element of `vc[i..i+4]` is computed in parallel

Single Instruction Multiple Data (SIMD)

- Scalar instruction: produces data in 1 cycle (to simplify)



- A SIMD instruction produces n data in 1 cycle (to simplify)



- SIMD instructions operate on so-called "vector" registers (rv_a, rv_b, rv_c)

Vector Instructions versus SIMD Instructions

Vector instructions :

- User-defined operation size
- No vector registers
- Implemented in old and some new supercomputers (SVE, Fugaku)

SIMD instructions:

- Fixed operation size for a given processor
- Presence of vector registers
- Implemented in most computers and current supercomputers (AVX, NEON)

In this session, we'll be focusing on SIMD instructions, as they are the most common in today's processor architectures.

SIMD and Vector Instruction Sets – Part 1

Date	ISA	Type	Micro-architecture	Registers Size
1997	MMX	SIMD	Intel Pentium	64-bit
1999	Altivec	SIMD	Apple+IBM PowerPC	128-bit
1999	SSE	SIMD	Intel Pentium III	128-bit
2000	SSE2	SIMD	Intel Pentium IV	128-bit
2004	SSE3	SIMD	Intel Pentium IV (Prescot)	128-bit
2005	NEONv1	SIMD	ARMv7	128-bit
2008	SSE4	SIMD	Intel Core 2 Duo	128-bit
2011	AVX	SIMD	Intel Sandy Bridge (Core i)	256-bit
2012	NEONv2	SIMD	ARMv8	128-bit
2013	AVX2	SIMD	Intel Haswell (Core iX)	256-bit
2016	AVX-512	SIMD	Intel Xeon Phi	512-bit
2017	SVE	Vector	ARMv8	–
2019	SVE2	Vector	ARMv9	–
2021	RVV	Vector	RISC-V	–

SIMD and Vector Instruction Sets – Part 2

Date	ISA	Type	Micro-architecture	Registers Size
1997	MMX	SIMD	Intel Pentium	64-bit
1999	Altivec	SIMD	Apple+IBM PowerPC	128-bit
1999	SSE	SIMD	Intel Pentium III	128-bit
2000	SSE2	SIMD	Intel Pentium IV	128-bit
2004	SSE3	SIMD	Intel Pentium IV (Prescot)	128-bit
2005	NEONv1	SIMD	ARMv7	128-bit
2008	SSE4	SIMD	Intel Core 2 Duo	128-bit
2011	AVX	SIMD	Intel Sandy Bridge (Core i)	256-bit
2012	NEONv2	SIMD	ARMv8	128-bit
2013	AVX2	SIMD	Intel Haswell (Core iX)	256-bit
2016	AVX-512	SIMD	Intel Xeon Phi	512-bit
2017	SVE	Vector	ARMv8	–
2019	SVE2	Vector	ARMv9	–
2021	RVV	Vector	RISC-V	–

- A large number of existing instruction sets
- Some are backward-compatible
  - AVX-512 → AVX → SSE → MMX
  - NEONv2 → NEONv1
- Most instruction sets are incompatible with each other
  - Each instruction set is written differently

Assembly Code

Example of single precision floating-point addition according to different ISAs:

x86 SSE

```
1 addps xmm, xmm
```

ARM NEON

```
1 FADD Vd.4S,Vn.4S,Vm.4S
```

x86 AVX

```
1 vaddps ymm, ymm, ymm
```

ARM SVE

```
1 FADDP Zresult.S, Pg.M, Zresult.S, Zop2.S
```

x86 AVX-512

```
1 vaddps zmm, k, zmm, zmm
```

RISC-V Vector extension

```
1 vadd.vv vd, vs2, vs1, vm
```

- As many different ways as there are instruction sets
- Assembly code is not portable :-(
- A major productivity brake

Intrinsic Functions: Example of Addition

x86 AVX

```
1 __m256 _mm256_add_ps (__m256 a, __m256 b);
```

x86 AVX-512

```
1 __m512 _mm512_mask_add_ps (__m512 src, __mmask16 k, __m512 a, __m512 b);
```

ARM NEON

```
1 float32x4_t vaddq_f32 (float32x4_t a, float32x4_t b);
```

ARM SVE

```
1 svfloat32_t svaddp_f32_x (svbool_t pg, svfloat32_t op1, svfloat32_t op2);
```

RISC-V Vector extension

```
1 vfloat32m1_t vfadd_vv_f32m1_m (vbool32_t mask, vfloat32m1_t maskedoff, vfloat32m1_t op1, vfloat32m1_t op2, size_t vl);
```

Intrinsic Functions

Definition: an intrinsic function is a function to which one (and only one) assembly instruction corresponds.

- As many different ways as there are instruction sets
- Code with intrinsic functions is not portable
- Better productivity than pure assembly programming
  - The compiler takes care of register allocation
  - Easy to interface with C and C++ code
  - Often the method chosen by embedded system developers

Work with the Compiler

- In some cases, the compiler can automatically vectorize the source code
  - Works when computation are regular and with `-O3` optimization level (GCC)
  - You can display a vectorization report to see which parts of the code have actually been vectorized
- The compiler often does not vectorize well
  - Either it does not vectorize at all
  - Or it vectorizes a little, but we could do much better
- It is possible to help the compiler with annotations (ex. OpenMP SIMD)
- In HPC, a lot of code uses the compiler + annotations
  - Supercomputers evolve fast
  - Relying on the compiler guarantees source code portability

Table of Contents

Source of Inspiration

This section of is strongly inspired by the excellent slides of
Pr. Lionel Lacassagne (Sorbonne University) for the same class.

Special thanks to him for sharing its work!

Strongly-typed Data Types

- C89 and before
  - — { `signed`, `unsigned` } × { `char` }, { `short`, `long` } × { `int` }, `float`, `double`
  - — ➔ `unsigned short`, `signed long int`, ...
  - — Major portability problem: type size can change depending on arch. and OS
- From C99 (`stdint.h`)
  - — Fixed-width integer types: `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`
  - — But the fixed-width floating-point data types are missing
- ARM choose to use the same philosophy
  - — `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `float16_t`, `float32_t`, `float64_t`
  - — ➔ Fixed-width floating-point data types are present!

Strongly-typed SIMD Data Types – Part 1

SIMD types use strong typing: `<type><size>x<number of lanes>_t`

- 64-bit registers
  - Signed integers: `int8x8_t`, `int16x4_t`, `int32x2_t`, `int64_t`,
  - Unsigned integers: `uint8x8_t`, `uint16x4_t`, `uint32x2_t`, `uint64_t`,
  - Floating-point numbers: `float16x4_t`, `float32x2_t`, `float64_t`
  - Polynomial: `poly8x8_t`, `poly16x4_t`
- 128-bit registers
  - Signed integers: `int8x16_t`, `int16x8_t`, `int32x4_t`, `int64x2_t`,
  - Unsigned integers: `uint8x16_t`, `uint16x8_t`, `uint32x4_t`, `uint64x2_t`,
  - Floating-point numbers: `float16x8_t`, `float32x4_t`, `float64x2_t`
  - Polynomial: `poly8x16_t`, `poly16x8_t`

Strongly-typed SIMD Data Types – Part 2

SIMD types use strong typing: `<type><size>x<number of lanes>_t`

The availability of the different types and their associated instructions depends on the architecture version ({ v7, v8 } × { AArch32, AArch64 }), their specialization (ARM v8.2 for `float16_t`) and the implementations by designers (Apple, Nvidia, Samsung, Qualcomm, Huawei, ...).

Type of Instructions

4 main types of SIMD instructions:

1. Initialization and memory access
   - `load`, `store`, `set`
2. Arithmetic and logic
   - `add`, `mul`, `div`, `shift`, ...
   - `andv`, `or`, `xor`, `not`
3. Comparison
   - $<$, $\leq$, $=$, $\neq$, $\geq$, $>$ (Fortran mnemonics: lt, le, eq, neq, ge, gt)
4. Shuffle & permutation
   - to reorganize values in a SIMD register, or in memory
   - because without it, certain computation would be inefficient in SIMD

Type of Instructions

4 main types of SIMD instructions:

1. Initialization and memory access
   - `load`, `store`, `set`
2. Arithmetic and logic
   - `add`, `mul`, `div`, `shift`, ...
   - `andv`, `or`, `xor`, `not`
3. Comparison
   - $<, \leq, =, \neq, \geq, >$ (Fortran mnemonics: lt, le, eq, neq, ge, gt)
4. Shuffle & permutation
   - to reorganize values in a SIMD register, or in memory
   - because without it, certain computation would be inefficient in SIMD

It is not required to know all the intrinsics but we need to be able to find the instruction we are looking for from the technical documentation.

Initialization – vdup_n / vmov_n

- Set of values (or variables)
  - C++ style: `T x = {...};`
  - Example: `uint32x4_t x = {1, 2, 3, 4};` ➔ `x = [1, 2, 3, 4]`
  - Possible with scalar variables of compatible type (in this case `uint32_t`)
- Unique value (or unique variable) – `vdup_n_T`, `vdupq_n_T`
  - `uint32_t u = 1;`
  - Example 64-bit: `uint32x2_t x = vdup_n_u32(1);` ➔ `x = [1, 1]`
  - Example 128-bit: `uint32x4_t x = vdupq_n_u32(u);` ➔ `x = [1, 1, 1, 1]`
- Unique value (or unique variable) – `vmov_n_T`, `vmovq_n_T`
  - `uint32_t u = 1;`
  - Example 64-bit: `uint32x2_t x = vmov_n_u32(1);` ➔ `x = [1, 1]`
  - Example 128-bit: `uint32x4_t x = vmovq_n_u32(u);` ➔ `x = [1, 1, 1, 1]`

Access to a Lane – `vset_lane` / `vget_lane`

- Set – `dst = vset_lane_T(val, src, i)`
  - Copy `src` into `dst` and write `val` in the `i`-lane of `dst`
  - To modify a register within a loop (efficiently)
- Get – `val = vget_lane_T(src, i)`
  - Extract the `i`-element
- Example

```
1 uint32x4_t x = (uint32x4_t) {10, 11, 12, 13}; // [10 11 12 13]
2 uint32x4_t y = vsetq_lane_u32(14, x, 3);      // [10 11 12 14]
3 uint32_t   t = vgetq_lane_u32(y, 2);          // 12
```

Memory Access – `vld1` / `vst1`

- LoaD – `vld1_T(addr)`, `vld1q_T(addr)`
- STore – `vst1_T(addr, x)`, `vst1q_T(addr, x)`

```
1 // 64-bit loads ('T8', 'T16' and 'T32' are data pointers, 'r<T>_64' are 64-bit registers)
2 uint8x8_t  r8_64  = vld1_u8 (T8 ); // [0, 1, 2, 3, 4, 5, 6, 7]_M
3 uint16x4_t r16_64 = vld1_u16(T16); // [01, 23, 45, 67]_M
4 uint32x2_t r32_64 = vld1_u32(T32); // [0123, 4567]_M
5 // --------------------------
6 // 128-bit stores ('T8', 'T16' and 'T32' are data pointers, 'r<T>_128' are 128-bit registers)
7 vst1q_u8 (T8 , r8_128 ); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F]_M
8 vst1q_u16(T16, r16_128); // [01, 23, 45, 67, 89, AB, CD, EF]_M
9 vst1q_u32(T32, r32_128); // [01234567, 89ABCDEF]_M
```

- The basic memory access instructions manipulate 1 set of elements, but there are also instructions that manipulate 2, 3 or even 4 sets of elements
  - For strided accesses 1:2, 1:3, 1:4
  - For conversions: Array-of-Struct - Struct-of-Array (AoS / SoA)

Display for Debug

- No working code without debugging!!
- Here is an example of printing SIMD register values
  - 8-bit elements & 128-bit SIMD register ➔ 16 elements

```
1 void display_r8_128(uint8x16_t r8_128, char *format) {
2     uint8_t T[16];              // declare and allocate array in memory
3     vst1q_u8(T, r8_128);        // write the vector register in memory
4     for(int i = 0; i < 16; i++) // for each element of the SIMD register
5       printf(format, T[i]);     // print the value of lane i
6 }
```

- SIMD register elements (= lanes) displayed in ascending order
  - As if the data were in memory
  - Same display on little and big endian architectures

Scalar Access to a Reg from Mem – `vld1_lane`

- `u = vld1_lane(p, v, i)`
  - Copy `v` in `u` and write the value pointed by `p` in the `i`-lane of `u`
  - $u \leftarrow v$, $u_i \leftarrow *p$

- Example
  - To simplify `T[i] = i;`
  - The index of cell `i` is the same with its contents

```
1 uint32_t T[4] = {0, 1, 2, 3};                 // T[ 0,  1,  2,  3]
2 uint32x4_t v = (uint32x4_t) {10, 11, 12, 13}; // v[10, 11, 12, 13]
3 uint32x4_t u;                                 // u[??, ??, ??, ??]
4 u = vld1q_lane_u32(T+0, v, 2);                // u[10, 11, 0, 13]
5 u = vld1q_lane_u32(T+2, v, 1);                // u[10, 2, 12, 13]
```

  - $u_2 \leftarrow T[0] = 0$
  - $u_1 \leftarrow T[2] = 2$

Splat from Memory – `vld1_dup`

- `u = vld1_dup(p)`
  - Broadcast over the entire `u` register, the value indicated by `p`
  - Close to `vdup_n(*p)`

- Example
  - To simplify `T[i] = i;`
  - The index of cell `i` is the same with its contents

```
1 uint32_t T[4] = {0, 1, 2, 3}; // T[0, 1, 2, 3]
2 uint32x4_t u;
3 u = vld1q_dup_u32(T+0);       // u[0, 0, 0, 0]
4 u = vld1q_dup_u32(T+2);       // u[2, 2, 2, 2]
```

Cast – `vreinterpret`

- Usage
  - To change the type of a SIMD variable (without changing its total size)
  - To apply low-level processing (bit by bit)
- Integer ⇔ integer
  - Sign change: `vreinterpret_i32_u32`, `vreinterpret_u32_i32`
  - Size change: `vreinterpret_u8_u32`, `vreinterpret_u32_u8`
- Integer ⇔ floating-point
  - Type change: `vreinterpret_f32_i32`, `vreinterpret_i32_f32`
- Free!
  - Only for the compiler, does not generate instruction
  - SIMD types are seen as 64- or 128-bit “packets” = weak typing

Split – vcombine / vget_low + vget_high

- Usage
  - To combine two 64-bit SIMD registers into one 128-bit SIMD register
  - To split a 128-bit SIMD register into two 64-bit SIMD registers

- Example

```
1 uint32x4_t w  = (uint32x4_t) {1, 2, 3, 4}; // [1 2 3 4]
2 uint32x2_t v0 = vget_low_u32(w);           // [1 2]
3 uint32x2_t v1 = vget_high_u32(w);          // [3 4]
4 uint32x4_t s  = vcombine_u32(v1, v0);      // [3 4 1 2]
```

- Permutation of left and right blocks (not efficient)

Splat / Duplication – `vdup_lane`

- Usage
  - To duplicate the value of a lane to the entire SIMD register
- Example

```
1 uint32x4_t x = (uint32x4_t) {10, 11, 12, 13}; // [10 11 12 13]
2 uint32x4_t y = vdupq_lane_u32(x, 1);          // [11 11 11 11]
```

- Utility: pattern for block multiplication

Binary Operations – `not` / `or` / `and` / `xor`

- Basic operators
  - NOT: `vmovn` (MOVe Not)
  - AND: `vand`
  - OR: `vorr` (OR Register)
  - XOR: `veor` (Exclusive OR)
- Advanced / specialized operators
  - OR NOT: `vorn` (OR Not)
  - AND NOT: `vbic` (BIt Clear)
  - REV: `vrbit` (Reverse BIT) inverts the bits of each element
  - XOR[2]: `veor3` *3-way exclusive OR*
  - `vbcax` *bit clear and exclusive OR*

Arithmetic Operations – `add` / `sub` / `mul` / `div`

- Classic arithmetic operations
  - ADD: `vadd` 8, 16, 32, 64
  - SUB: `vsub` 8, 16, 32, 64
  - NEG: `vneg` 8, 16, 32, 64
  - MUL: `vmul` 8, 16, 32, 64
  - DIV: `vdiv` 16, 32, 64 (only for floats)
- Specialized arithmetic operations
  - MUL: `vmul_lane(a, b, i)`: $a \times b_i$
  - MUL: `vmul_n(v, s)`: multiplication by a scalar value

Arithmetic Operations – Fused Multiply Add

- FMA ARMv7 = `vmla` / `vmls`
  - `vmla(a, b, c)` $a + b \times c$ (MuL-Add)
  - `vmls(a, b, c)` $a - b \times c$ (MuL-Sub)
  - Integer: 8, 16, 32 – Float: 16, 32, 64
- FMA ARMv8 (IEEE-754), so only F_{16}, F_{32}, F_{64}
  - `vfma(a, b, c)`: $a + b \times c$
  - `vfms(a, b, c)`: $a - b \times c$
  - `vfma_lane(a, b, c, i)`: $a + b \times c_i$
  - `vfms_lane(a, b, c, i)`: $a - b \times c_i$
  - `vfma_n(a, b, s)`: $a + b \times s$ ($\leftarrow$ scalar)
  - `vfms_n(a, b, s)`: $a - b \times s$ ($\leftarrow$ scalar)
- Extension (reduction)
  - `vdot`: scalar product TODO

Saturated Arithmetic Operations – Integers

1. General case of arithmetic: size change (C and mathematics)
   - Addition $n + n \rightarrow n + 1$ bits so $2n$ bits
   - Multiplication: $n \times n \rightarrow 2n$ bits
   - $\Rightarrow$ Loss of parallelism ($\div 2$)
   - The size change prevents the compiler from vectorizing integer codes
   - Example `(a + b + c + d) / 4`
2. Special case #1: without size change
   - Addition: $n + n \rightarrow n$ bits because info on missing addition holdback
   - Multiplication: $n \times n \rightarrow 2n$ because operands on n bits
   - Signal and image processing
   - Bad example `(a / 4 + b / 4 + c / 4 + d / 4)` (loss of precision)
3. Special case #2: with only a few holdbacks
   - Often the case with audio
   - Computations without loss of parallelism + saturation of results that exceed (internal holdback test)

Saturated Arithmetic Ops – `vqadd` / `vqsub`

- Only for addition (`vqadd`) and subtraction (`vqsub`)
  - — Unsigned saturated arithmetic `vqadd_u`, `vqsub_u` $[0, 2^n - 1]$
  - — Signed saturated arithmetic `vqadd_s`, `vqsub_s` $[-2^{n-1}, 2^{n-1} - 1]$

```
1  uint8x8_t u = (uint8x8_t) {1, 2, 3, 4, 5, 6, 7, 8};
2                        // [  1   2   3   4   5   6   7   8]
3  u = vqadd_u8(u, u); // [  2   4   6   8  10  12  14  16]
4  u = vqadd_u8(u, u); // [  4   8  12  16  20  24  28  32]
5  u = vqadd_u8(u, u); // [  8  16  24  32  40  48  56  64]
6  u = vqadd_u8(u, u); // [ 16  32  48  64  80  96 112 128]
7  u = vqadd_u8(u, u); // [ 32  64  96 128 160 192 224 255] <- sat(128 + 128) = sat(256) = 255
8  u = vqadd_u8(u, u); // [ 64 128 192 255 255 255 255 255]
9  u = vqadd_u8(u, u); // [128 255 255 255 255 255 255 255]
10 u = vqadd_u8(u, u); // [255 255 255 255 255 255 255 255]
```

Rounding – vrnd

- 4 rounding modes (IEEE-754 compliant)
  - — `vrnd`: rounding to zero
  - — `vrndn`: rounding to *nearest*
  - — `vrndm`: rounding to *minus* infinity
  - — `vrndp`: rounding to *plus* infinity
- Available for all 3 registers types and 2 sizes:
  - — { `f16`, `f32`, `f64` } × { 64-bit, 128-bit }

```
1 float32x4_t x;
2 float32x4_t r;
3 x = (float32x4_t)      { -1.75f, -1.25f, 1.25f, 1.75f };
4 r = vrndq_f32 (x);  // [ -1.00f  -1.00f  1.00f  1.00f ] // zero
5 r = vrndnq_f32(x);  // [ -2.00f  -1.00f  1.00f  2.00f ] // nearest
6 r = vrndmq_f32(x);  // [ -1.00f  -1.00f  2.00f  2.00f ] // minus infinity
7 r = vrndpq_f32(x);  // [ -2.00f  -2.00f  1.00f  1.00f ] // plus infinity
```

Conversion – `vcvt`

- Conversion from `float` to `int` and `uint`, 4 rounding modes (IEEE-754)
  - `vcvt_s_f`: convert with rounding to zero
  - `vcvtn_s_f` convert with rounding to *nearest*
  - `vcvtm_s_f` convert with rounding to *minus* infinity
  - `vcvtp_s_f` convert with rounding to *plus* infinity
- Available for all 3 registers types and 2 sizes:
  - { `f16`, `f32`, `f64` } × { 64-bit, 128-bit } (to signed and unsigned integers)
- Reverse conversion from `int` to `float`: `vcvt_f_s`

```
1 float32x4_t x; int32x4_t c;
2 // float -> int
3 x = (float32x4_t)         { -1.75f, -1.25f, 1.25f, 1.75f };
4 c = vcvtq_s32_f32 (x); // [ -1      -1      1      1     ] // zero
5 c = vcvtnq_s32_f32(x); // [ -2      -1      1      2     ] // nearest
6 c = vcvtmq_s32_f32(x); // [ -1      -1      2      2     ] // minus infinity
7 c = vcvtpq_s32_f32(x); // [ -2      -2      1      1     ] // plus infinity
8 // int -> float
9 x = vcvtmq_f32_s32(c); // [ -1.0f   -1.0f   2.0f   2.0f  ] // zero
```

Ordering instructions – `min` / `max` – Part 1

- Two missing scalar instructions
  - In scalar mode, use a control structure: `if(a<b) m=a; else m=b`
  - The extremum is computed by subtraction and comparison of the sign of the result: `s=a-b; if(s<0) m=a; else m=b;`
  - On simple processors, the control structure generates a pipeline stalls
  - But on complex processors, there is a conditional `move` (= `cmove`)
- In SIMD
  - The extremum is computed without any control structure
  - SIMD version more is efficient

Ordering instructions – `min` / `max` – Part 2

Extraction of min and max scalars from 2 arrays

- Array version (v1) (most compact)

```
1 for (int i = 0; i < n; i++) {
2   if (A[i] < B[i]) {
3     Min[i] = A[i]; Max[i] = B[i];
4   } else {
5     Min[i] = B[i]; Max[i] = A[i];
6   }
7 }
```

- Version with ternary conditions (v2)

```
1 for (int i = 0; i < n; i++) {
2   uint8_t a = A[i];
3   uint8_t b = B[i];
4   uint8_t m = (a < b) ? a : b; // better without if
5   uint8_t M = (a < b) ? b : a; // 'cmov' instr.
6   Min[i] = m; Max[i] = M;
7 }
```

- SIMD version

```
1 for (int i = 0; i < n; i += 16) {
2   uint8x16_t a = vld1q_u8(A + i);
3   uint8x16_t b = vld1q_u8(&B[i]);
4   uint8x16_t m = vminq_u8(a, b);
5   uint8x16_t M = vmaxq_u8(a, b);
6   vst1q_u8(Min + i, m);
7   vst1q_u8(&Max[i], M);
8 }
```

— SIMD code is easy to write from the scalar version 2

Comparison instructions – vc

- SIMD intrinsic names are derived from Fortran mnemonics
  - For integers and floats
  - `vclt <` LT: Less Than
  - `vcle <=` LE: Less or Equal
  - `vceq ==` EQ: EQual
  - `vcge >=` GE: Greater or Equal
  - `vcgt >` GT: Greater Than
- Extension to unary (monoadic) operators
  - Avoids the use of a second register initialized to zero
  - For integers signed and floats
  - `vcltz` < 0
  - `vclez` ≤ 0
  - `vceqz` $= 0$
  - `vcgez` ≥ 0
  - `vcgtz` > 0

Design Pattern Selection – `vbsl`

- Considering the `min` computations
  - Assumption for the example: `min` and `max` don't exist
  - Coding without control structure but with comparison instructions
    `c=a<b; m=(a&c)|(b&!c)`
  - With scalar macros `c=CMPLT(a,b); m=OR(AND(a,c),AND(b,NOT(c)));`
  - And if `andnot` exists: `c=CMPLT(a,b); m=OR(AND(a,c),ANDNOT(b,c));`
  - Assumption comparisons return `0x00` or `0xff`
- In SIMD
  - `c=vclt_u8(a,b); m=vorr_u8(vand_u8(a,c),vbic_u8(b,c)));`
- NEON dedicated instruction for the selection: *Bit SeLect* (`vbsl`)
  - If $c = 1$, then $m = v_T$ else $m = v_F$ ➔ `m = vbsl(c, vT, vF);`

Web References

- Neon Intrinsics Reference
  https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics
- Armv8 Instruction Set Architecture: https://developer.arm.com/docs/ddi0596/c/simd-and-floating-point-instructions-alphabetic-order
- Coding for NEON
  - Part 1: Load and Stores https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/coding-for-neon---part-1-load-and-stores
  - Part 2: Dealing With Leftovers https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/coding-for-neon---part-2-dealing-with-leftovers
  - Part 3: Matrix Multiplication https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/coding-for-neon---part-3-matrix-multiplication
  - Part 4: Shifting Left and Right https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/coding-for-neon---part-4-shifting-left-and-right
  - Part 5: Rearranging Vectors https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/coding-for-neon---part-5-rearranging-vectors

Table of Contents

SIMD wrappers – Definition

Definition: a SIMD wrapper is a library (C or C++) that call intrinsic functions or assembler. The SIMD wrapper provides a single programming interface for the various instruction sets.

- Improves source code expressiveness
  - Possibility of using traditional C operators (`+`, `-`, `*`, `/`, ...)
- Source code is portable
  - You only need to write the source code once, and it will compile on several different architectures

Let's take a look at the SIMD "MIPP" wrapper. It's a good compromise between efficiency and portability.

“My Intrinsics++” (MIPP)

- MyIntrinsicsPlusPlus (MIPP), is a code is written in C++ and based on SIMD intrinsic function calls
- Used in several scientific codes to benefit from SIMD instructions
- Open source code: https://github.com/aff3ct/MIPP
- Exhaustively tested in a continuous integration pipeline (guaranteed to be bug-free, or nearly so...)
- Other wrappers exist, but the mechanics are identical
  - Mastering MIPP = facilitating understanding of intrinsic functions
  - Mastering MIPP = getting to grips with other SIMD wrappers faster

MIPP: Hello World!

3 SIMD Wrapper: My Intrinsics++ (MIPP)

```cpp
void add_vector(const float *A, const float *B, float *C, int size) {
  // N elements per SIMD register (static = know at compilation time)
  constexpr int N = mipp::N<float>();
  // vectorized loop
  for (int i = 0; i < size; i += N) {
    mipp::Reg<float> rA = &A[i];   // SIMD read in memory
    mipp::Reg<float> rB = &B[i];   // SIMD read in memory
    mipp::Reg<float> rC = rA + rB; // SIMD addition
    rC.store(&C[i]);               // SIMD write in memory
  }
}
```

MIPP: Hello World!


```
1  void add_vector(const float *A, const float *B, float *C, int size) {
2    // N elements per SIMD register (static = know at compilation time)
3    constexpr int N = mipp::N<float>();
4    // vectorized loop
5    for (int i = 0; i < size; i += N) {
6      mipp::Reg<float> rA = &A[i];   // SIMD read in memory
7      mipp::Reg<float> rB = &B[i];   // SIMD read in memory
8      mipp::Reg<float> rC = rA + rB; // SIMD addition
9      rC.store(&C[i]);               // SIMD write in memory
10   }
11 }
```

- **The loop pitch is N!** (register cardinal)

MIPP: Hello World!


```
1  void add_vector(const float *A, const float *B, float *C, int size) {
2    // N elements per SIMD register (static = know at compilation time)
3    constexpr int N = mipp::N<float>();
4    // vectorized loop
5    for (int i = 0; i < size; i += N) {
6      mipp::Reg<float> rA = &A[i];   // SIMD read in memory
7      mipp::Reg<float> rB = &B[i];   // SIMD read in memory
8      mipp::Reg<float> rC = rA + rB; // SIMD addition
9      rC.store(&C[i]);               // SIMD write in memory
10   }
11 }
```

- **The loop pitch is N!** (register cardinal)
- MIPP works at the level of **vector registers**
  - A `mipp::Reg` can be considered as a REAL register in the processor
  - MIPP is written in C++, and the code between the chevrons is used to determine the type of elements inside the vector registers (here `float`)

Section 3.1

Review of the Main Operations

Registers and Memory

3 SIMD Wrapper: My Intrinsics++ (MIPP)

Defining a register from a constant

```
1 mipp::Reg<float> r_cst = 12.f; // r_cst = [12.f, 12.f, 12.f, 12.f]
```

Registers and Memory

3 SIMD Wrapper: My Intrinsics++ (MIPP)

Defining a register from a constant

```
1 mipp::Reg<float> r_cst = 12.f; // r_cst = [12.f, 12.f, 12.f, 12.f]
```

Read data from memory to a register

```
1 float array[4] = {0.f, 1.f, 2.f, 3.f};
2 mipp::Reg<float> r_vals = array; // r_cst = [0.f, 1.f, 2.f, 3.f]
```

Registers and Memory

3 SIMD Wrapper: My Intrinsics++ (MIPP)

Defining a register from a constant

```
1 mipp::Reg<float> r_cst = 12.f; // r_cst = [12.f, 12.f, 12.f, 12.f]
```

Read data from memory to a register

```
1 float array[4] = {0.f, 1.f, 2.f, 3.f};
2 mipp::Reg<float> r_vals = array; // r_cst = [0.f, 1.f, 2.f, 3.f]
```

Write data from a register to memory

```
1 mipp::Reg<float> r_vals = {4.f, 5.f, 6.f, 7.f};
2 float array[4] = {0.f, 0.f, 0.f, 0.f};
3 r_vals.store(array); // array = [4.f, 5.f, 6.f, 7.f]
```

Registers and Memory

3 SIMD Wrapper: My Intrinsics++ (MIPP)

Defining a register from a constant

```
1 mipp::Reg<float> r_cst = 12.f; // r_cst = [12.f, 12.f, 12.f, 12.f]
```

Read data from memory to a register

```
1 float array[4] = {0.f, 1.f, 2.f, 3.f};
2 mipp::Reg<float> r_vals = array; // r_cst = [0.f, 1.f, 2.f, 3.f]
```

Write data from a register to memory

```
1 mipp::Reg<float> r_vals = {4.f, 5.f, 6.f, 7.f};
2 float array[4] = {0.f, 0.f, 0.f, 0.f};
3 r_vals.store(array); // array = [4.f, 5.f, 6.f, 7.f]
```

Display register value on standard output

```
1 mipp::Reg<float> r_vals = {4.f, 5.f, 6.f, 7.f};
2 mipp::dump<float>(r_vals.r); // display "[4.f, 5.f, 6.f, 7.f]" on the screen
```

Usual Arithmetic Operations

- MIPP supports most common arithmetic operations
  - `mipp::add(...)`, `mipp::sub(...)`, `mipp::mul(...)`, `mipp::div(...)`
  - You can also use the operators `+`, `-`, `*`, `/` between two `mipp::Reg`

```
1 mipp::Reg<float> r_a = {0.f, 1.f, 2.f, 3.f};
2 mipp::Reg<float> r_b = {4.f, 5.f, 6.f, 7.f};
3 mipp::Reg<float> r_c1 = mipp::add(r_a, r_b); // r_c1 = [4.f, 6.f, 8.f, 10.f]
4 mipp::Reg<float> r_c2 = r_a + r_b;           // r_c2 = [4.f, 6.f, 8.f, 10.f]
```

- Using operator overloading is strictly identical to calling `mipp::add`
- It's one of the great advantages of SIMD wrappers ;-)

Specialized Arithmetic Operations – FMA

- SIMD instruction sets are designed for computational efficiency
- Several specific instructions are implemented
  - The best-known operation is *Fused Multiply and Add* (FMA): $d = a \times b + c$
    - This is a recurring operation in many HPC applications
    - In MIPP: `mipp::fmadd`, `mipp::fnmadd`, `mipp::fmsub`, `mipp::fnmsub`
    - No C++ operator for the FMA, so you have to call MIPP functions...

```
1 mipp::Reg<float> r_a = {0.f, 1.f, 2.f, 3.f};
2 mipp::Reg<float> r_b = {4.f, 5.f, 6.f, 7.f};
3 mipp::Reg<float> r_c = {1.f, 2.f, 1.f, 2.f};
4 mipp::Reg<float> r_c1 = mipp::fmadd(r_a, r_b, r_c); // r_c1 = [1.f, 7.f, 13.f, 16.f]
5 mipp::Reg<float> r_c2 = r_a * r_b + r_b;            // r_c2 = [1.f, 7.f, 13.f, 16.f]
```

- Please note that lines 4 and 5 are not equivalent
  - Line 4 executes 1 instruction, while line 5 executes 2
  - Line 4 has a single rounding, while line 5 has two

Specialized Arithmetic Operations – Other

- There are many other specialized operations
  - The square root, the inverse of the square root
  - Small matrix multiplication for AI (= tensor, Intel AMX)
  - 32-bit CRC calculation
  - And so on
- These specialized functions vary according to the instruction set
  - Instruction sets are very heterogeneous, and choices are made according to the targeted applications
- MIPP supports part of these specialized operations
  - See the documentation: https://github.com/aff3ct/MIPP/blob/master/README.md

Masks and Logical Operations

MIPP supports most common logic operations:

- `mipp::cmpeq`, `mipp::cmpneq`, `mipp::cmpge`, `mipp::cmpgt`, `mipp::cmple`, `mipp::cmplt`
- You can also use the operators `==`, `!=`, `>=`, `>`, `<=`, `<`
- The use of logical operations on vector registers has the effect of returning **a mask-type register!**

```
1 mipp::Reg<float> r_a = {0.f, 5.f, 2.f, 7.f};
2 mipp::Reg<float> r_b = {4.f, 1.f, 6.f, 3.f};
3 mipp::Msk<mipp::N<float()> r_m = r_a > r_b; // r_m = [0, 1, 0, 1]
```

- A mask vector register contains the information *true* or *false*
- Here we use `mipp::N<float>()` to retrieve the number of elements contained in a data vector register (`mipp::Reg<float>`)

Bitwise Operations

MIPP supports most common bitwise operations:

- `mipp::andb`, `mipp::orb`, `mipp::xorb`, `mipp::lshift`, `mipp::rshift`, `mipp::notb`, ...
- You can also use the operators `&`, `|`, `^`, `<<`, `>>`, `~`

Note that bitwise operations apply to both data vector registers (`mipp::Reg`) and mask vector registers (`mipp::Msk`).

Ternary Condition or the *Blend* in Vector Form

- As you may have already noticed, there is no SIMD condition
- However, it is possible to implement a ternary condition (or select)

```
1 int res = (val1 > val2) ? val1 : val2; // res = max(val1, val2)
```

Ternary Condition or the *Blend* in Vector Form

- As you may have already noticed, there is no SIMD condition
- However, it is possible to implement a ternary condition (or select)

```
1 int res = (val1 > val2) ? val1 : val2; // res = max(val1, val2)
```

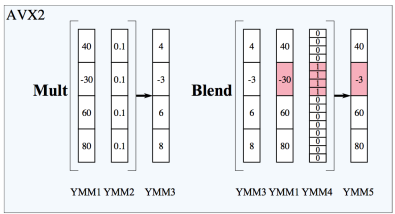
- This behavior can be mimicked in SIMD

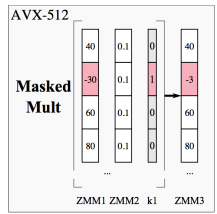
```
1 mipp::Reg<int> r_vals1 = {12, 1, 23, 4};
2 mipp::Reg<int> r_vals2 = {3, 89, 24, 0};
3 mipp::Msk<mipp::N<int>()> r_m = r_vals1 > r_vals2;
4 mipp::Reg<int> r_res1 = mipp::blend(r_vals1, r_vals2, r_m);
5 // r_res1 = {12, 89, 24, 4}
6 mipp::Reg<int> r_res2 = mipp::blend(r_vals1, r_vals2, r_vals1 > r_vals2);
7 // r_res2 = {12, 89, 24, 4}
```

— Lines 3-4 or 6 are strictly equivalent
— Line 6 (computation of `r_res2`) is a compressed version

Masked SIMD Instruction Sets

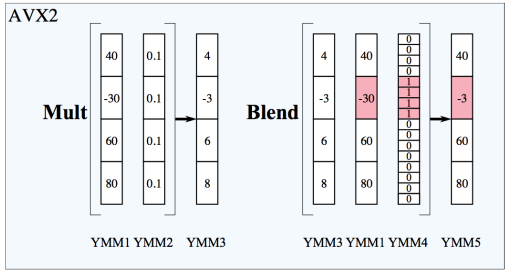
- Combine an arithmetic operation and a *blend* in one instruction
  - Available with AVX-512, SVE and RVV
  - Can't be written with a `mipp::blend`
  - If the instruction set doesn't support it, the masked arithmetic operation is emulated (implementation call the `mipp::blend`)

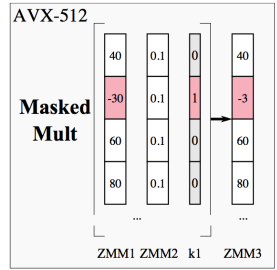




Masked SIMD Instruction Sets – Implementation

3 SIMD Wrapper: My Intrinsics++ (MIPP)






```
1 mipp::Reg<double> r_zmm1 = {40, -30, 60, 80};
2 mipp::Reg<double> r_zmm2 = {0.1, 0.1, 0.1, 0.1};
3 mipp::Msk<mipp::N<double>()> r_k1 = {0, 1, 0, 0}
4 mipp::Reg<double> r_zmm3 = mipp::mask<double,mipp::mul>(r_k1, r_zmm1, r_zmm1, r_zmm2);
5 // r_zmm3 = [40, -3, 60, 80]
```

Shuffles and Permutations

Most SIMD (and NOT vector) instruction sets include instructions for reordering elements within a vector register.

- Being able to shuffle elements implies knowing the number of elements (here we assume 4 floating elements in a vector register)

```
1 uint32_t ids[4] = {3, 2, 1, 0};
2 mipp::Reg<float> cmask = mipp::cmask(ids);
3 mipp::Reg<float> r_vals = {10.f, 20.f, 30.f, 40.f};
4 mipp::Reg<float> r_rvals = mipp::shuff(r_vals, cmask);
5 // r_rvals = [40.f, 30.f, 20.f, 10.f]
```

As the implementation of `mipp::shuff` is often quite costly in terms of assembler instructions, this operation should only be used if necessary.

Gather and Scatter

The *gather* operation is used to load elements of an array of values into memory from an index vector register to initialize a vector register.

```
1 float mem[16] = {0.0f, 10.1f, 20.2f, 30.3f, 40.4f, 50.5f, /*...*/ 150.15f};
2 mipp::Reg<int> r_idx = {0, 5, 12, 2};
3 mipp::Reg<float> r_vals = mipp::gather<float>(mem, r_idx);
4 // r_vals = [0.0f, 50.5f, 120.12f, 20.2f]
```

The *scatter* operation writes the elements of a vector register to memory. Addresses are determined by an index vector register.

```
1 float mem[16] = {0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, /*...*/ 0.0f};
2 mipp::Reg<int> r_idx = {0, 5, 12, 2};
3 mipp::Reg<float> r_vals = {0.12f, 1.12f, 2.12f, 3.12f};
4 mipp::scatter<float>(mem, r_idx, r_vals);
5 // mem = [0.12f, 0.0f, 3.12f, 0.0f, 0.0f, 1.12f, /*...*/ 0.0f]
```

Reductions or Horizontal Operations

A reduction (or horizontal operation) is an operation involving elements of the same vector register.

- Some examples of reductions
  - The sum of elements within a single vector register
  - The product of elements within the same vector register
  - Find the minimum/maximum element within the same vector register
  - And so on
- MIPP support most standard reductions (`mipp::hadd`, `mipp::hmul`, `mipp::hmin`, `mipp::hmax`, `mipp::testz`)

```
1 mipp::Reg<int> r_vals = {1, 2, 3, 4};
2 int sum = mipp::hadd<int>(r_vals); // sum = 10
```

Section 3.2

Vectorization Strategies

Tail Loop


```
1  // assume that "size = 18"
2  void add_vector(const float *A, const float *B, float *C, int size) {
3    constexpr int N = mipp::N<float>(); // assume that "N = 4"
4    int vec_loop_end = (size / N) * N; // "vec_loop_end = 16"
5
6    // vectorized loop
7    for (int i = 0; i < vec_loop_end; i += N) {
8      mipp::Reg<float> rA = &A[i];
9      mipp::Reg<float> rB = &B[i];
10     mipp::Reg<float> rC = rA + rB;
11     rC.store(&C[i]);
12   }
13
14   // tail loop (scalar ops) for the remaining elements (i = 16 et i = 17)
15   for (int i = vec_loop_end; i < size; i++)
16     C[i] = A[i] + B[i];
17 }
```

Masked Loads and Stores

3 SIMD Wrapper: My Intrinsics++ (MIPP)

```cpp
// assume that "size = 18"
void add_vector(const float *A, const float *B, float *C, int size) {
  constexpr int N = mipp::N<float>(); // assume that "N = 4"
  mipp::Reg<int> r_limit = size; // r_limit = [18, 18, 18, 18]
  mipp::Reg<int> r_count = {0, 1, 2, 3};
  // vectorized loop
  for (int i = 0; i < size; i += N) {
    mipp::Msk<mipp::N<int>()> r_m = r_count < r_limit;
    // masked read, puts a 0 in the register if the mask is 0
    mipp::Reg<float> rA = mipp::maskzld(r_m, &A[i]);
    mipp::Reg<float> rB = mipp::maskzld(r_m, &B[i]);
    mipp::Reg<float> rC = rA + rB;
    // masked write, does not write to memory if mask element is 0
    mipp::maskst(r_m, &C[i], rC);
    r_count += N; // increment the counter by N
  }
}
```

Q&A

Thank you for listening!
Do you have any questions?