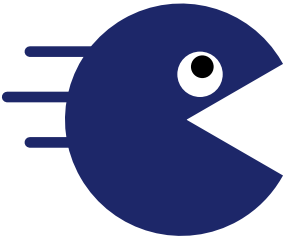


Single Instruction Multiple Data

AVX-512 Instruction Set Architecture

Polytech Sorbonne – EI5-SE – Calcul haute performance (EPU-F9-IHP)

Adrien Cassagne

November 10, 2025

Table of Contents

Session Objectives

- Vector and SIMD programming models
  - Origins and definitions
  - Several programming levels
- AVX-512 Instruction Set Architecture (ISA)
  - Presentation and philosophy
  - Intrinsic functions

Table of Contents

2 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

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 vadd c(1..10000), a(1..10000), b(1..10000)
```

- The `vadd` 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
```

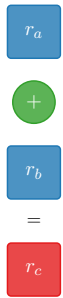
Example of SIMD instructions:

```
1  loop:
2    vload va, [p_a] # read 4 elements
3    vload vb, [p_b] # read 4 elements
4    vadd vc, va, vb # add 4 elements
5    vstore [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
```

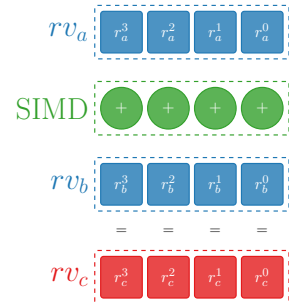
- The `vadd`, `vload` and `vstore` instructions perform operations on 4 elements at a time
  - 4 times fewer instructions (potential acceleration of 4)
  - The number of elements in a vector register = the # lanes (here 4)
  - Each element of `vc[i..i+4]` is computed in parallel

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 (RVV, 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	128-bit to 2048-bit
2019	SVE2	Vector	ARMv9	128-bit to 2048-bit
2021	RVV	Vector	RISC-V	128-bit to 65536-bit

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	128-bit to 2048-bit
2019	SVE2	Vector	ARMv9	128-bit to 2048-bit
2021	RVV	Vector	RISC-V	128-bit to 65536-bit

- 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 chosen method by embedded system developers

Work with the Compiler

- In some cases, the compiler can automatically vectorize the source code
  - Works when computations 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

Short History

2013 SIMD ISA proposed by Intel
- — Successor of AVX (256-bit) and SSE (128-bit) but on **512-bit**

2016 First implementation in the **Intel Xeon Phi x200** (Knights Landing)

2017 First time in CPUs: **Skylake-SP** and **Skylake-X** (6th generation of Intel Core processors)

2021 **Cancelled in Intel consumer-grade CPUs** since Alder Lake (12th generation of Intel Core processors)

2022 Adopted by the latest AMD architectures (**Zen 4** in 2022 and **Zen 5** in 2024)

2022 Now **the standard on x86 server-grade CPUs** (from Intel Sapphire Rapids in 2023 and Zen 4/5 AMD EPYC in 2022)

2023 AVX10 is presented by Intel: No new feature, simplifies ISA detection

Specificity (1)

- 32 × 512-bit vector registers: `ZMM0-ZMM31`
- A lot of parallelism:
  - 64 × 8-bit elements (`int8_t`, `uint8_t`)
  - 32 × 16-bit elements (`int16_t`, `uint16_t`, `bf16_t`)
  - 16 × 32-bit elements (`int32_t`, `uint32_t`, `float32_t`)
  - 8 × 64-bit elements (`int64_t`, `uint64_t`, `float64_t`)
- 8 × 64-bit (or 32-bit if no `BW`) mask registers: `K0-K7`
  - 32 × 64-bit on server CPUs: `K0-K31`
- Organized in extensions
  - One mandatory extension, called “Foundation”: `F`
  - A lot of other optional extensions: `CD`, `ER`, `PF`, `4VNNIW`, `4FMAPS`, `VL`, `DQ`, `BW`, `IFMA`, `VBMI`, `VNNI`, `VPOPCNTDQ`, `VBMI2`, `BITALG`, `VP2INTERSECT`, `GFNI`, `VPCLMULQDQ`, `VAES`

Specificity (2)

- EVEX encoding with maximum 5 operands (including 2 optional ones)

```
VOP ZMMA {KA}{ZMMB}, ZMMC, ZMMD
```

  - `ZMMA`, `ZMMB`, `ZMMC`, `ZMMD` ∈ `ZMM0-31` and `KA` ∈ `K0-7`
  - If present, `KA` encodes the use of opmask register for conditional processing and updates to destination
  - If present, `ZMMB` enable merging when `KA` is true (requires `KA`)

- 2 partitions with AVX-512 in Dalek:
  - `az4-mixed`: 8 × AMD Ryzen 9 7945HX (Zen 4)
    - Supported extensions: `F`, `CD`, `VL`, `DQ`, `BW`, `IFMA`, `VBMI`, `VBMI2`, `VPOPCNTDQ`, `BITALG`, `VNNI`, `VPCLMULQDQ`, `GFNI`, `VAES`, `BF16`
    - AVX-512 ISA but 256-bit SIMD ALUs
  - `az5-a890m`: 4 × AMD Ryzen AI 9 HX 370 (Zen 5)
    - Supported extensions: Same as Zen 4 + `VP2INTERSECT`
    - AVX-512 ISA and 512-bit SIMD ALUs

Extensions – F, CD, ER, PF

Introduced with Xeon Phi x200 (**2016**, Knights Landing) and after in Xeon Scalable (**2017**, Skylake-SP "Purley"), with `ER` and `PF` being specific to Knights Landing & Knights Mill (**2017**):

`F` **Foundation Instructions** – expands most 32-bit and 64-bit based AVX instructions with the EVEX coding scheme to support 512-bit registers, operation masks, parameter broadcasting, and embedded rounding and exception control, implemented by Knights Landing and Skylake Xeon

`CD` **Conflict Detection Instructions** – efficient conflict detection to allow more loops to be vectorized, implemented by Knights Landing and Skylake-X

`ER` **Exponential and Reciprocal Instructions** – exponential and reciprocal operations designed to help implement transcendental operations, implemented by Knights Landing

`PF` **Prefetch Instructions** – new prefetch capabilities, implemented by Knights Landing

Extensions – 4VNNIW, 4FMAPS

Introduced with and specific to Knights Mill (**2017**):

`4VNNIW` **Vector Neural Network Instructions Word** variable precision – vector instructions for deep learning, enhanced word, variable precision

`4FMAPS` **Fused Multiply Accumulation Packed Single** precision – vector instructions for deep learning, floating point, single precision

Extensions – VL, DQ, BW, IFMA, VBMI

Introduced with Skylake-SP/X (**2017**, **2019**) and Cannon Lake (**2018**):

- `VL` **Vector Length Extensions** – extends most AVX-512 operations to also operate on XMM (128-bit) and YMM (256-bit) registers
- `DQ` **Doubleword and Quadword Instructions** – adds new 32-bit and 64-bit AVX-512 instructions
- `BW` **Byte and Word Instructions** – extends AVX-512 to cover 8-bit and 16-bit integer operations

Introduced with Cannon Lake (**2018**):

- `IFMA` **Integer Fused Multiply Add** – fused multiply add of integers using 52-bit precision
- `VBMI` **Vector Bit Manipulation Instructions** – adds vector byte permutation instructions which were not present in AVX-512BW

Extensions – VNNI, VPOPCNTDQ, VBMI2, BITALG

Introduced with Cascade Lake (**2019**):

VNNI **Vector Neural Network Instructions** – vector instructions for deep learning.

Introduced with Knights Mill (**2017**) and Ice Lake (**2021**):

VPOPCNTDQ **Vector population count instruction**

Introduced with Ice Lake (**2021**):

VBMI2 **Vector Bit Manipulation Instructions 2** – byte/word load, store and concatenation with shift.

BITALG **AVX-512 Bit Algorithms** – byte/word bit manipulation instructions expanding VPOPCNTDQ

Ext. – VP2INTERSECT, GFNI, VPCLMULQDQ, VAES

Introduced with Tiger Lake (**2020**):

VP2INTERSECT **Vector Pair Intersection to a Pair of Mask Registers**

Introduced with Ice Lake (**2021**):

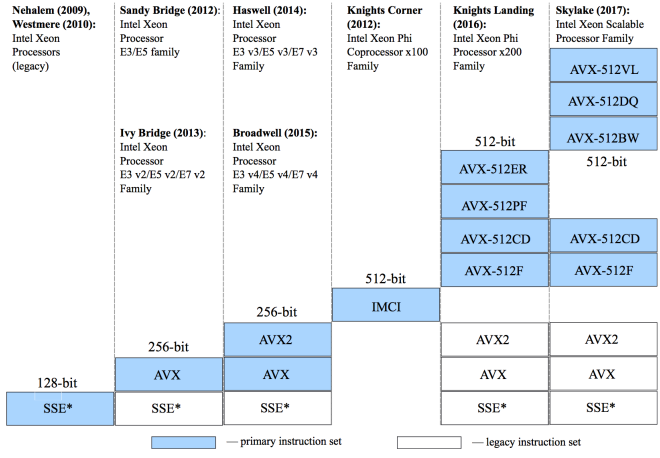
GFNI **Galois Field New Instructions**

VPCLMULQDQ **Vector Carry-Less Multiplication of Quadwords**

VAES **AES instructions**

Intel SIMD ISA Evolution

3 AVX-512 Presentation



From https://colfaxresearch.com/skl-avx512/.

AVX-512 Extensions per Architecture

3 AVX-512 Presentation

Designer	Microarchitecture	Year	Support Level																	
			F	CD	ER	PF	BW	DQ	VL	FP16	IFMA	VBMI	VBMI2	BITALG	VPOPCNTDQ	VP2INTERSECT	4VNNIW	4FMAPS	VNNI	BF16
Intel	Knights Landing	2016	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘
	Knights Mill	2017	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✔	✘	✔	✔	✘	✘
	Skylake (server)	2017	✔	✔	✘	✘	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘
	Cannon Lake	2018	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘
	Cascade Lake	2019	✔	✔	✘	✘	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘	✔	✘
	Cooper Lake	2020	✔	✔	✘	✘	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘	✔	✔
	Tiger Lake	2020	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔	✘	✘	✔	✘
	Rocket Lake	2021	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✘	✘	✘	✔	✘
	Alder Lake	2021	✔	✔	✘	✘	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✘	✘	✔	✔
	Ice Lake (server)	2021	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✘	✘	✘	✔	✘
	Sapphire Rapids	2023	✔	✔	✘	✘	✔	✔	✔	✔	✔	✔	✔	✔	✔	✘	✘	✘	✔	✔
AMD	Zen 4	2022	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✘	✘	✘	✔	✔
	Zen 5	2024	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔	✘	✘	✔	✔
Centaur	CHA		✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘

From https://en.wikichip.org/wiki/x86/avx512_vnni.

Table of Contents

4 AVX-512 SIMD Programming

Strongly-typed Data Types (1)

- 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
- `float` is sometimes referred as `float32_t` (IEEE 754)
- `double` is sometimes referred as `float64_t` (IEEE 754)

Strongly-typed Data Types (2)

In this class, we introduce the following notations:

- Signed integers
  - `int8_t` → `i8`
  - `int16_t` → `i16`
  - `int32_t` → `i32`
  - `int64_t` → `i64`
- Unsigned integers
  - `uint8_t` → `u8`
  - `uint16_t` → `u16`
  - `uint32_t` → `u32`
  - `uint64_t` → `u64`
- Floating-point integers
  - `float32_t` → `f32` (also known as `float`)
  - `float64_t` → `f64` (also known as `double`)
- Boolean (= 1 bit) → `b`

Registers and Intrinsic Type (1)

- Vector registers `ZMM0-ZMM31` are not typed: Set of 512-bit
- Corresponding intrisic types:

`__mm512` For single (32-bit) precision floating-point numbers
`__mm512d` For double (64-bit) precision floating-point numbers
➔ Element type is **strongly-typed**

`__mm512i` For signed and unsigned integers (8- to 64-bit)
➔ Element size is not encoded in the type
➔ Element type is **loosely-typed**

- Mask registers `K0-K7` are boolean (32- or 64-bit)
- Corresponding intrisic types **contain the number of elements**:

`__mmask8` Mask 8 elements (for `float64_t`, `int64_t` and `uint64_t`)
`__mmask16` Mask 16 elements (for `float32_t`, `int32_t` and `uint32_t`)
`__mmask32` Mask 32 elements (for `bf16_t`, `int16_t` and `uint16_t`)
`__mmask64` Mask 64 elements (for `int8_t` and `uint8_t`, AVX-512 `BW`)

511 256	255 128	127 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30
ZMM31	YMM31	XMM31

Registers and Intrinsic Type (2)

- **Intrinsic types are NOT registers**
  - When working with intrinsics the registers allocation is made by the the compiler
  - As much as possible the compiler tries to allocate intrinsics types to register
  - When too much intrinsic variables are used, the compiler spills them

```
1  #include <immintrin.h>
2
3  __mm512  vec1_var = _mm512_setzero();       // likely to be mapped to ZMM0 -- [0,...,0]^{f32}_{16}
4  __mm512d vec2_var = _mm512_setzero_pd();    // likely to be mapped to ZMM1 -- [0,...,0]^{f64}_{8}
5  __mm512i vec3_var = _mm512_setzero_epi32(); // likely to be mapped to ZMM2 -- [0,...,0]^{i32}_{16}
6
7  __mmask8  mask1_var = 0x0F;       // likely to be mapped to K0 -- [1,1,1,1,0,0,0,0]^{b}_{8}
8  __mmask16 mask2_var = 0xAAAA;     // likely to be mapped to K1 -- [1,0,1,0,...,1,0]^{b}_{16}
9  __mmask32 mask3_var = 0xFFFFFFFF; // likely to be mapped to K2 -- [1,1,1,1,...,1,1]^{b}_{32}
```

- AVX-512 instrinsic types and functions are defined in the `immintrin.h` header

Instructions – Introduction

4 main types of SIMD instructions:

1. Initialization and memory access
   - `load`, `store`, `set`
2. Arithmetic and logic
   - `add`, `mul`, `div`, `shift`, ...
   - `and`, `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, some computations 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.

Instructions – Intrinsic Mnemonics (1)

Let's suppose classic binary operations (+, -, ×, /, AND, OR, ...), generally there are three possible type of intrinsic function signatures:

1. `VEC_T _mmVECSIZE_OPCODE_DATATYPE(VEC_T a, VEC_T b)`
2. `VEC_T _mmVECSIZE_maskz_OPCODE_DATATYPE(MSK_T k, VEC_T a, VEC_T b)`
3. `VEC_T _mmVECSIZE_mask_OPCODE_DATATYPE(VEC_T s, MSK_T k, VEC_T a, VEC_T b)`

- `VEC_T` ∈ `__mm512`, `__mm512d`, `__mm512i`, `__mm256`, `__mm256d`, `__mm256i`, `__mm128`, `__mm128d`, `__mm128i`,
- `MSK_T` ∈ `__mmask8`, `__mmask16`, `__mmask32`, `__mmask64`
- `VECSIZE` ∈ ∅ (= 128 bits), `256` (= 256 bits), `512` (= 512 bits)
- `DATATYPE` ∈ `pd` (= `f64`), `ps` (= `f32`), `epi64` (= `i64`), `epu64` (= `u64`), `epi32` (= `i32`), `epu32` (= `u32`), `epi16` (= `i16`), `epu16` (= `u16`), `epi8` (= `i8`), `epu8` (= `u8`), `si512`, `si256` and `si128` (for bitwise operations)

Instructions – Intrinsic Mnemonics (2)

Let's suppose classic binary operations (+, -, ×, /, AND, OR, ...), considering the first possible type of intrinsic function signature:

1. `VEC_T _mmVECSIZE_OPCODE_DATATYPE(VEC_T a, VEC_T b)`

Addition on single precision floating-point and 512-bits: $\boldsymbol{r} = \boldsymbol{a} + \boldsymbol{b}$

- `VEC_T` is `__mm512`
- `VECSIZE` is `512` (= 512 bits)
- `DATATYPE` is `ps` (= `f32`)
- `OPCODE` is `add`, no surprise

```
1  __mm512 a, b, r; // a = [1,...,1]_16^f32, b = [2,...,2]_16^f32, r = [?,...,?]_16^f32
2  r = _mm512_add_ps(a, b); // r = [3,...,3]_16^f32
```

➔ `VEC_T`, `VECSIZE` and `DATATYPE` need to be compatible!

Instructions – Intrinsic Mnemonics (3)

Let's suppose classic binary operations (+, -, ×, /, AND, OR, ...), considering the second possible type of intrinsic function signature:

2. `VEC_T _mmVECSIZE_maskz_OPCODE_DATATYPE(MSK_T k, VEC_T a, VEC_T b)`

Masked to zero addition on single precision floating-point and 512-bits:

$$r_i = \begin{cases} a_i + b_i, & \text{if } k_i = 1 \\ 0, & \text{otherwise.} \end{cases}$$

- `VEC_T`, `VECSIZE`, `DATATYPE` and `OPCODE` are the same as for case 1.
- `MSK_T` is `__mmask16`

```
1  __mm512 a, b, r; // a = [1,...,1]_16^f32 , b = [2,...,2]_16^f32 , r = [?,...,?]_16^f32
2  __mmask16 = 0xAAAA; // k = [1,0,...,1,0]_16^b
3  r = _mm512_maskz_add_ps(k, a, b); // r = [3,0,...,3,0]_16^f32
```

Instructions – Intrinsic Mnemonics (4)

Let's suppose classic binary operations (+, -, ×, /, AND, OR, ...), considering the third possible type of intrinsic function signature:

3. `VEC_T _mmVECSIZE_mask_OPCODE_DATATYPE(VEC_T s, MSK_T k, VEC_T a, VEC_T b)`

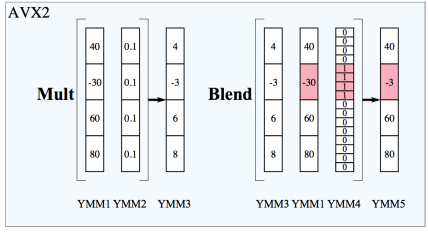
Masked addition on single precision floating-point and 512-bits:

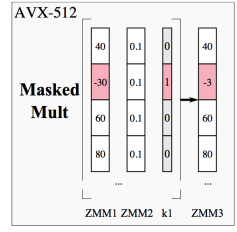
$$r_i = \begin{cases} a_i + b_i, & \text{if } k_i = 1 \\ s_i, & \text{otherwise.} \end{cases}$$

- `VEC_T`, `VECSIZE`, `DATATYPE`, `OPCODE` and `MSK_T` are the same as for case 2.

```
1  __mm512 a, b, r; // a = [1,...,1]_16^f32, b = [2,...,2]_16^f32, r = [?,...,?]_16^f32
2  __mmask16 = 0xAAAA; // k = [1,0,...,1,0]_16^b
3  __mm512 s; // s = [4,...,4]_16^f32
4  r = _mm512_mask_add_ps(s, k, a, b); // r = [3,4,...,3,4]_16^f32
```

Masking – Example and Comparison





From https://colfaxresearch.com/skl-avx512/.

Legacy AVX2 solution:

```
1 __mm256d YMM1, YMM2, YMM3, YMM4, YMM5;
2 YMM3 = _mm256_mul_pd(YMM1, YMM2);
3 YMM5 = _mm256_blendv_pd(YMM3, YMM1, YMM4);
```

➔ 5 vec. regs + 2 instr.

Modern AVX-512 solution:

```
1 __mm512d ZMM1, ZMM2, ZMM3;
2 __mmask8 k1;
3 ZMM3 = _mm512_mask_mul_pd(ZMM1, k1, ZMM1, ZMM2);
```

➔ 3 vec. regs + 1 mask reg + 1 instr.

Intrinsics – Initialization (1)

- To zeros
  - OPCODE ← setzero
  - No maskz and mask variants

```
// __m512d _mm512_setzero_pd ();
__m512d r = _mm512_setzero_pd(); // r = [0,...,0]_8^f64
```

- Broadcast a scalar value
  - OPCODE ← set1
  - maskz and mask variants sometime available

```
1 // __m512i _mm512_set1_epi32 (int a);
2 __m512i r1 = _mm512_set1_epi32(-2);  // r1 = [-2,...,-2]_16^i32
3
4 __mmask16 k = 0x00FF;                          //  k = [1,1,...,1, 0, 0,..., 0]_16^b
5 // __m512i _mm512_mask_set1_epi32 (__m512i src, __mmask16 k, int a);
6 __m512i r2 = _mm512_mask_set1_epi32(r1, k, 5); // r2 = [5,5,...,5,-2,-2,...,-2]_16^i32
```

Intrinsics – Initialization (2)

- Set of independent values
  - `OPCODE` ← `set` or `setr`
  - No `maskz` and `mask` variants

```
// __m512i _mm512_set_epi64 (__int64 e7, __int64 e6, __int64 e5, __int64 e4,
//                           __int64 e3, __int64 e2, __int64 e1, __int64 e0);
__m512i r = _mm512_set_epi64(7, 6, 5, 4, -3, -2, -1, 0); // r = [0,-1,-2,-3,4,5,6,7]_8^i64
```

```
// __m512i _mm512_setr_epi64 (__int64 e0, __int64 e1, __int64 e2, __int64 e3,
//                            __int64 e4, __int64 e5, __int64 e6, __int64 e7);
__m512i r = _mm512_setr_epi64(0, -1, -2, -3, 4, 5, 6, 7); // r = [0,-1,-2,-3,4,5,6,7]_8^i64
```

Intrinsics – Get an Element

- Extract a specific element
  - OPCODE ← `extract`
  - Does not exist in AVX-512, only exists in AVX2 and for integers...

```
// int32_t _mm256_extract_epi8  (__m256i a, const int index);
// int32_t _mm256_extract_epi16 (__m256i a, const int index);
// int32_t _mm256_extract_epi32 (__m256i a, const int index);
// int64_t _mm256_extract_epi64 (__m256i a, const int index);
```

  - Workaround: Use the memory...
- Get the first element
  - OPCODE ← `cvtsd`, `cvtss` or `cvtsi512`

```
// double _mm512_cvtsd_f64     (__m512d a);
// float  _mm512_cvtss_f32     (__m512  a);
// int    _mm512_cvtsi512_si32 (__m512i a);
```

Intrinsics – Set an Element

- Set an element in the vector
  - OPCODE ← `set1` with the `mask` variant

```
1 unsigned int index = 2;
2 __m512i r1 = _mm512_setr_epi64(0, -1, -2, -3, 4, 5, 6, 7); // r1 = [0,-1,-2,-3,4,5,6,7]_8^i64
3 // __m512i _mm512_mask_set1_epi64 (__m512i src, __mmask8 k, __int64 a)
4 r1 = _mm512_mask_set1_epi64(r1, 1 << index, 10);   // r1 = [0,-1,10,-3,4,5,6,7]_8^i64
```

  - `_mm512_mask_set1_ps` and `_mm512_mask_set1_pd` intrinsics are missing :-(
  - Workaround: Interpret float as integer and use `mask_set1_epi` intrinsics...

Intrinsics – Memory Load (1)

- Load a vector from memory
  - OPCODE ← `loadu` or `load`

```
1 int64_t dataptr[8] = {0, 1, 2, 3, 4, 5, 6, 7};
2 // __m512i _mm512_loadu_epi64 (void const* mem_addr);
3 __m512i r = _mm512_loadu_epi64(dataptr); // r = [0,1,2,3,4,5,6,7]_8^i64
```

➔ Unaligned load are supported

➔ If `dataptr` is aligned on 64 bytes, the cost is similar to `load`

```
1 int64_t dataptr1[8] = {0, 1, 2, 3, 4, 5, 6, 7};
2 // __m512i _mm512_load_epi64 (void const* mem_addr);
3 __m512i r1 = _mm512_load_epi64(dataptr1); // likely to produce a segfault
4
5 int64_t *dataptr2 = aligned_alloc(64, 8 * sizeof(int64_t)); // from C11
6 for (int i = 0; i < 8; i++) dataptr2[i] = 7 - i;
7 __m512i r2 = _mm512_load_epi64(dataptr2); // r2 = [7,6,5,4,3,2,1,0]_8^i64
```

➔ Data alignment on a 64 bytes address is required

Intrinsics – Memory Load (2)

- Load a vector from memory but **masked**
  - OPCODE ← `loadu` or `load`

```
1 double dataptr[4] = {0, 1, 2, 3};
2 __m512d s = _mm512_set1_pd(9); // s = [9,9,9,9,9,9,9,9]_8^f64
3 __mmask8 k = 0x0F; // k = [1,1,1,1,0,0,0,0]_8^b
4 // __m512d _mm512_mask_loadu_pd (__m512d src, __mmask8 k, void const* mem_addr);
5 __m512d r = _mm512_mask_loadu_pd(s, k, dataptr); // r = [0,1,2,3,9,9,9,9]_8^f64
```

  - ➔ No segfault! When the element is masked the HW does not perform the load
  - Also work with `maskz` variant
  - Similar behavior with `mask[z]_load` intrinsics

Intrinsics – Memory Load (3)

- Gather elements from memory
  - OPCODE ← i32gather

```
1 double dataptr[32] = {31, 30, 29, 28, 27, 26, 25, 24,
2                       23, 22, 21, 20, 19, 18, 17, 16,
3                       15, 14, 13, 12, 11, 10,  9,  8,
4                        7,  6,  5,  4,  3,  2,  1,  0};
5 __m256i vindex = _mm256_setr_epi32(2, 4, 12, 15, 16, 22, 29, 30);
6 // __m512d _mm512_i32gather_pd (__m256i vindex, void const* base_addr, int scale)
7 __m512d r = _mm512_i32gather_pd(vindex, dataptr, 8); // r = [29,27,19,16,15,9,2,1]_8^f64
```

  - The `scale` parameter is generally the element size, here `sizeof(double) == 8`
  - Also work with `mask` variant
  - Badly implemented in current architectures, the loads are serialized, **try to avoid gather when possible**

Intrinsics – Memory Store (1)

- Store a vector register into the memory
  - OPCODE ← `storeu` or `store`

```
1 double dataptr[8];
2 __m512d a = _mm512_setr_pd(0,1,2,3,4,5,6,7); // a = [0,1,2,3,4,5,6,7]_8^f64
3 // void _mm512_storeu_pd (void* mem_addr, __m512d a);
4 _mm512_storeu_pd(dataptr, a); // dataptr = [0,1,2,3,4,5,6,7]_8^f64
```

  - `store` variant for 64B aligned data also supported

- Store a vector register into the memory with **masking**
  - OPCODE ← `storeu` or `store` (does not support `maskz` variant)

```
1 double dataptr[8] = {9, 9, 9, 9, 9, 9, 9, 9};
2 __m512d a = _mm512_setr_pd(0,1,2,3,4,5,6,7); // a = [0,1,2,3,4,5,6,7]_8^f64
3 __mmask8 k = 0xF0;                          // k = [0,0,0,0,1,1,1,1]_8^b
4 // void _mm512_mask_storeu_pd (void* mem_addr, __mmask8 k, __m512d a);
5 _mm512_mask_storeu_pd(dataptr, k, a);  // dataptr = [9,9,9,9,4,5,6,7]_8^f64
```

Intrinsics – Memory Store (2)

- Compress store a vector register into the memory
  - OPCODE ← `compressstoreu`

```
1 int64_t dataptr[4]; __mmask8 k = 0xAA; // k = [1,0,1,0,1,0,1,0]_8^b
2 __m512i a = _mm512_setr_epi64(0,1,2,3,4,5,6,7); // a = [0,1,2,3,4,5,6,7]_8^i64
3 // void _mm512_mask_compressstoreu_epi64 (void* base_addr, __mmask8 k, __m512i a);
4 _mm512_mask_compressstoreu_epi64(dataptr, k, a); // dataptr = [0,2,4,6]_4^i64
```

  - Compress store is a very common pattern in applications
- Scatter vector elements into the memory
  - OPCODE ← `i32scatter` (`mask` variant also exists)

```
1 int64_t dataptr[32];
2 __m512i a = _mm512_setr_epi64(0,1,2,3,4,5,6,7); // a = [0,1,2,3,4,5,6,7]_8^i64
3 __m256i vindex = _mm256_setr_epi32(2, 4, 12, 15, 16, 22, 29, 30);
4 // void _mm512_i32scatter_epi64 (void* base_addr, __m256i vindex, __m512i a, int scale)
5 _mm512_i32scatter_epi64(dataptr, vindex, a); // dataptr = [?, ?, 0, ?, 1, ?, ?, ?,
6                                              //            ?, ?, ?, ?, 2, ?, ?, 3,
7                                              //            4, ?, ?, ?, ?, ?, 5, ?,
8                                              //            ?, ?, ?, ?, ?, 6, 7, ?]_32^i64
```

Intrinsics – Masks Load and Store

- Load a mask from memory
  - OPCODE ← load

```
1 __mmask8  _load_mask8  (__mmask8*  mem_addr);
2 __mmask16 _load_mask16 (__mmask16* mem_addr);
3 __mmask32 _load_mask32 (__mmask32* mem_addr);
4 __mmask64 _load_mask64 (__mmask64* mem_addr);
```

- Store a mask into memory
  - OPCODE ← store

```
1 void _store_mask8  (__mmask8*  mem_addr, __mmask8  a);
2 void _store_mask16 (__mmask16* mem_addr, __mmask16 a);
3 void _store_mask32 (__mmask32* mem_addr, __mmask32 a);
4 void _store_mask64 (__mmask64* mem_addr, __mmask64 a);
```

- Memory pointers are always aligned on 16 bytes!

➔ No alignment issue with mask loads and stores

Intrinsics – Display for Debug

- No working code without **debugging**!
- Here is an example of printing SIMD register values
  - 8-bit elements & 512-bit SIMD register ➔ 64 elements

```
1 void display_i8_512(__m512i ri8_512, char *format) {
2   int8_t T[64];                    // declare and allocate array on the stack memory
3   _mm512_storeu_epi8(T, ri8_512); // write the vector register in memory
4   for (int i = 0; i < 64; i++)     // for each element of the SIMD register
5     printf(format, T[i]);          // print the value of element i
6 }
```

  - SIMD register elements are displayed in ascending order
- Never cast a SIMD vector register into a pointer!

```
1 void display_i8_512(__m512i ri8_512, char *format) {
2   int8_t* T = (int8_t*)ri8_512; // cast a register into an address, **FORBIDDEN**!!
3   for (int i = 0; i < 64; i++)   // for each element of the SIMD register
4     printf(format, T[i]);        // print the value of element i, **MAY SEGFAULT**
5 }
```

Intrinsics – Cast

- Change the type of a SIMD vector variable (without changing its total size)
  - `OPCODE` ← `cast`
- Integer ⇔ floating-point

```
1 __m512i _mm512_castps_si512 (__m512  a); // f32 -> integer
2 __m512  _mm512_castsi512_ps (__m512i a); // integer -> f32
3 __m512i _mm512_castpd_si512 (__m512d a); // f64 -> integer
4 __m512d _mm512_castsi512_pd (__m512i a); // integer -> f64
```

- Float ⇔ float

```
1 __m512  _mm512_castpd_ps (__m512d a); // f64 -> f32
2 __m512d _mm512_castps_pd (__m512  a); // f32 -> f64
```

- Integer ⇔ integer
  - No cast available since integers are loosely-typed in `__m512i` type

➔ **Free!** No additional instruction in asm code, this is only for compiler checks

Intrinsics – Split Low and High

- Split low and high part of a SIMD vector register
  - OPCODE ← `extract`

```
1  __m512i a = _mm512_setr_epi32(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);
2  // a = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]_16^i32
3  __m512d ad = _mm512_castsi512_pd(a); // ad = [?,?,?,?,?,?,?,?]_8^f64
4
5  // === GET LOW ===
6  // __m256d _mm512_extractf64x4_pd (__m512d a, int imm8);
7  __m256d ld = _mm512_extractf64x4_pd(ad, 0); // ld = [?,?,?,?]_4^f64
8  __m256i li = _mm256_castpd_si256(ld);       // li = [0,1,2,3,4,5,6,7]_8^i32
9
10 // === GET HIGH ===
11 // __m256d _mm512_extractf64x4_pd (__m512d a, int imm8);
12 __m256d hd = _mm512_extractf64x4_pd(ad, 1); // hd = [?,?,?,?]_4^f64
13 __m256i hi = _mm256_castpd_si256(hd);       // hi = [8,9,10,11,12,13,14,15]_8^i32
```

  - Extract register quarters to go from 512 bits to 128 bits (`extractf32x4`)
  - `mask` and `maskz` variants available

Intrinsics – Combine Low and High

- Combine two half registers to create an full length register
  - `OPCODE` ← `insert`

```
1  _m256i li = _mm256_setr_epi32(0,1, 2, 3, 4, 5, 6, 7); // li = [0,1, 2, 3, 4, 5, 6, 7]_8^i32
2  _m256i hi = _mm256_setr_epi32(8,9,10,11,12,13,14,15); // hi = [8,9,10,11,12,13,14,15]_8^i32
3
4  _m256d ld = _mm256_castsi256_pd(li) // ld = [?,?,?,?]_4^f64
5  _m256d hd = _mm256_castsi256_pd(hi) // hd = [?,?,?,?]_4^f64
6
7  _m512d fld = _mm512_castpd256_pd512(ld); // fld = [?,?,?,?,?,?,?,?]_8^f64
8
9  // __m512d _mm512_insertf64x4 (__m512d a, __m256d b, int imm8)
10 __m512d rd = _mm512_insertf64x4(fld, hd, 1); // rd = [?,?,?,?,?,?,?,?]_8^f64
11 __m512i ri = _mm512_castpd_si512(ld); // ri = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]_16^i32
```

  - Not possible to combine 4 register quarters in one instruction
  - `mask` and `maskz` variants available

Intrinsics – Bitwise Operations on Vectors (1)

- AND: OPCODE ← and (_mm512_and_si512)
- OR: OPCODE ← or (_mm512_or_si512)
- Exclusive OR: OPCODE ← xor (_mm512_xor_si512)
- NOT: Not available in AVX-512 :-(

```
1 __m512i _mm512_not_si512_emu(__m512i a) {
2 __m512i allones = _mm512_set1_epi32(-1); // allones = 0xFFFF...F_512b
3 return _mm512_xor_si512(a, allones);
4 }
```

- AND NOT: OPCODE ← andnot (_mm512_andnot_si512)

```
1 __m512i a = _mm512_setr_epi8(0xAA, /* ..., */ 0xAA); // a = 0xAAAA...A_512b
2 __m512i b = _mm512_setr_epi8(0x55, /* ..., */ 0x55); // b = 0x5555...5_512b
3 // __m512i _mm512_andnot_si512 (__m512i a, __m512i b);
4 __m512i r = _mm512_andnot_si512(a,b); // r = NOT(a) AND b = 0x5555...5_512b AND 0x5555...5_512b
5                                       // r = 0x5555...5_512b
```

➔ mask and maskz variants available

Intrinsics – Bitwise Operations on Vectors (2)

- SHIFT LEFT: `OPCODE` ← `slli` or `sllv`

```
1 // constant shift immediate for all the elements
2 __m512i _mm512_slli_epi16 (__m512i a, unsigned int imm8);
3 __m512i _mm512_slli_epi32 (__m512i a, unsigned int imm8);
4 __m512i _mm512_slli_epi64 (__m512i a, unsigned int imm8);
5 // different shift values depending on the element
6 __m512i _mm512_sllv_epi16 (__m512i a, __m512i count);
7 __m512i _mm512_sllv_epi32 (__m512i a, __m512i count);
8 __m512i _mm512_sllv_epi64 (__m512i a, __m512i count);
```

- SHIFT RIGHT: `OPCODE` ← `srli`, `srlv`, `srai` or `srav`
  - Logical and arithmetic right shifts are available
- POPCOUNT: `OPCODE` ← `popcnt`

```
1 __m512i a = _mm512_setr_epi8(0, 1, 2, 3, 4, 5, 6, 7, /*...,*/ 63);
2 // __m512i _mm512_popcnt_epi8 (__m512i a);
3 __m512i r = _mm512_popcnt_epi8(a); // r = [0,1,1,2,1,2,2,3, /*...,*/ 6]_64^i8
```

➔ `mask` and `maskz` variants are available

Intrinsics – Bitwise Operations on Masks (1)

- AND: OPCODE ← kand (_mm512_kand)
- OR: OPCODE ← kor (_mm512_kor)
- Exclusive OR: OPCODE ← kxor (_mm512_kxor)
- AND NOT: OPCODE ← kandn (_mm512_kandn)
- NOT: OPCODE ← knot (_mm512_knot)
- SHIFT LEFT: OPCODE ← kshiftli

```
1 __mmask8  _kshiftli_mask8  (__mmask8  a, unsigned int count);
2 __mmask16 _kshiftli_mask16 (__mmask16 a, unsigned int count);
3 __mmask32 _kshiftli_mask32 (__mmask32 a, unsigned int count);
4 __mmask64 _kshiftli_mask64 (__mmask64 a, unsigned int count);
```

Intrinsics – Bitwise Operations on Masks (2)

- SHIFT RIGHT: OPCODE ← kshiftri

```
1 __mmask8  _kshiftri_mask8  (__mmask8  a, unsigned int count);
2 __mmask16 _kshiftri_mask16 (__mmask16 a, unsigned int count);
3 __mmask32 _kshiftri_mask32 (__mmask32 a, unsigned int count);
4 __mmask64 _kshiftri_mask64 (__mmask64 a, unsigned int count);
```

— No right arithmetic shift on masks

- KORTESTZ: OPCODE ← kortestz

```
1 __mmask8 k1 = 0xAA; // k1 = [1,0,1,0,1,0,1,0]_8^b
2 __mmask8 k2 = 0x55; // k2 = [0,1,0,1,0,1,0,1]_8^b
3 // int _mm512_kortestz (__mmask16 k1, __mmask16 k2);
4 // => set ZF flag if the returned val is equal to 0
5 int r1 = _mm512_kortestz(k1, k2); // r1 = [0, ..., 0, 1,1,1,1,1,1,1,1]_32^b
6 int r2 = _mm512_kortestz(k1, k1); // r2 = [0, ..., 0, 1,0,1,0,1,0,1,0]_32^b
```

— Determine if all the elements are masked ➔ Classic stop criterion

Intrinsics – Bitwise Operations on Masks (3)

- KUNPCK: OPCODE ← kunpack

```
1 __mmask16 _mm512_kunpackb (__mmask16 a, __mmask16 b);
2 __mmask32 _mm512_kunpackw (__mmask32 a, __mmask32 b);
3 __mmask64 _mm512_kunpackd (__mmask64 a, __mmask64 b);
```

```
1  __m256 zero = _mm256_setzero_ps();
2  __m256 lo = _mm256_setr_ps(0,-1,2,-3,4,-5,6,-7), hi = _mm256_setr_ps(8,-9,10,-11,12,-13,14,-15);
3
4  __mmask8 mlo = _mm256_cmp_epi32_mask(lo, zero, _MM_CMPINT_LE); // mlo = [1,1,0,1,0,1,0,1]_8^b
5  __mmask8 mhi = _mm256_cmp_epi32_mask(hi, zero, _MM_CMPINT_LE); // mhi = [0,1,0,1,0,1,0,1]_8^b
6
7  __m256d lod = _mm256_castps_pd(lo); __m256d hid = _mm256_castps_pd(hi);
8  __m512d flod = _mm512_castpd256_pd512(lod), lodhid = _mm512_insertf64x4(flod, hid, 1);
9  __mmask16 mlo2 = (__mmask16)mlo; // mlo2 = [1,1,0,1,0,1,0,1,0,0,0,0,0,0,0,0]_16^b
10 __mmask16 mhi2 = (__mmask16)mhi; // mhi2 = [0,1,0,1,0,1,0,1,0,0,0,0,0,0,0,0]_16^b
11
12 __m512 lohi; __mmask16 mlohi;
13 lohi = _mm512_castpd_ps(lodhid); // lohi  = [0,-1,2,-3,4,-5,6,-7,8,-9,10,-11,12,-13,14,-15]_16^f32
14 mlohi = mm512_kunpackb(a,b);     // mlohi = [1, 1,0, 1,0, 1,0, 1,0, 1, 0,  1, 0,  1, 0,  1]_16^b
```

Intrinsics – Arithmetic Operations (1)

- Classic arithmetic operations
  - ADD: `OPCODE` ← `add` (i8-64, u8-64, f16-64)
  - SUB: `OPCODE` ← `sub` (i8-64, u8-64, f16-64)
  - MUL: `OPCODE` ← `mul` (i32, u32, f16-64), `mullo` (i16-64), `mulhi` (i16, u16)
  - DIV: `OPCODE` ← `div` (f16-64)

- Fused Multiply and Add (FMA)
  - FMA: `OPCODE` ← `fmadd` (f16-64) – $\boldsymbol{r} = \boldsymbol{a} \times \boldsymbol{b} + \boldsymbol{c}$
  - FNMA: `OPCODE` ← `fnmadd` (f16-64) – $\boldsymbol{r} = -(\boldsymbol{a} \times \boldsymbol{b}) + \boldsymbol{c}$
  - FMS: `OPCODE` ← `fmsub` (f16-64) – $\boldsymbol{r} = \boldsymbol{a} \times \boldsymbol{b} - \boldsymbol{c}$
  - FNMS: `OPCODE` ← `fnmsub` (f16-64) – $\boldsymbol{r} = -(\boldsymbol{a} \times \boldsymbol{b}) - \boldsymbol{c}$
  - FMAS `OPCODE` ← `fmaddsub` (f16-64) –

$$r_i = \begin{cases} a_i \times b_i + c_i, & \text{if } i \text{ is even} \\ a_i \times b_i - c_i, & \text{otherwise.} \end{cases}$$

➔ `mask` and `maskz` variants are available

Intrinsics – Arithmetic Operations (2)

- Dot Product Accumulate 2: OPCODE $\leftarrow$ dpwssd – $r_i^{i32} = s_i^{i32} + \sum_{j=i\times 2}^{i\times 2+2} a_j^{i16} \times b_j^{i16}$

```
1 __m512i s = _mm512_set1_epi32(1); // s = [1,...,1]^i32_16
2 __m512i a = _mm512_setr_epi16(0,1,2,3,4,5,6,7,/* ...,*/ 31); // a = [0,1,2,3,4,5,6,7,...,31]^i16_32
3 __m512i b = _mm512_setr_epi16(0,1,2,3,4,5,6,7,/* ...,*/ 31); // b = [0,1,2,3,4,5,6,7,...,31]^i16_32
4 // __m512i _mm512_dpwssd_epi32 (__m512i src, __m512i a, __m512i b);
5 __m512i r = _mm512_dpwssd_epi32(s, a, b); // r = [1+ 0*0+1*1, 1+ 2*2+3*3, ..., 1+ 30*30+31*31]^i32_16
6                                           // r = [1+       1, 1+      13, ..., 1+        1861]^i32_16
7                                           // r = [         2,        14, ...,          1862]^i32_16
```

- Dot Product Accumulate 4: OPCODE $\leftarrow$ dpbusd – $r_i^{i32} = s_i^{i32} + \sum_{j=i\times 4}^{i\times 4+4} a_j^{i8} \times b_j^{i8}$

```
1 __m512i s = _mm512_set1_epi32(1); // s = [1,...,1]^i32_16
2 __m512i a = _mm512_setr_epi8(0,1,2,3,4,5,6,7,/* ...,*/ 63); // a = [0,1,2,3,4,5,6,7,...,63]^i8_64
3 __m512i b = _mm512_setr_epi8(0,1,2,3,4,5,6,7,/* ...,*/ 63); // b = [0,1,2,3,4,5,6,7,...,63]^i8_64
4 // __m512i _mm512_dpbusd_epi32 (__m512i src, __m512i a, __m512i b);
5 __m512i r = _mm512_dpbusd_epi32(s, a, b);
6 // r = [1+ 0*0+1*1+2*2+3*3 , 1+ 4*4+5*5+6*6+7*7, ..., 1+ 60*60+61*61+62*62+63*63]^i32_16
7 // r = [1+              14,  1+             126, ..., 1+                  15 134]^i32_16
8 // r = [                15,                 127, ...,                  15 135]^i32_16
```

Intrinsics – Arithmetic Operations (3)

- DPA2 and DPA4 in Vector Neural Network Instructions extension (`VNNI`)
  - Nowadays, **peak performance indicator** on CPU
- DPA2 on modern VNNI architectures
  - Cycles Per Instruction (CPI) $= 0.5$ ➔ 2 instructions per cycle
  - Per instruction: $16_{\text{32-bit elmt}} \times (2_{\text{mul}} + 2_{\text{add}}) = 64$ operations
  - Total number of operations per cycle: $64/0.5 =$ **128**
- DPA4 on modern VNNI architectures
  - Cycles Per Instruction (CPI) $= 0.5$ ➔ 2 instructions per cycle
  - Per instruction: $16_{\text{32-bit elmt}} \times (4_{\text{mul}} + 4_{\text{add}}) = 128$ operations
  - Total number of operations per cycle: $128/0.5 =$ **256**

➔ Crazy high number of arithmetic operations per cycle

Intrinsics – Arithmetic Operations (4)

- Some unary operations
  - Absolute value: `OPCODE` ← `abs` (i8-64, f16-64)
  - Square root: `OPCODE` ← `sqrt` (f16-64)
  - Reciprocal of square root: `OPCODE` ← `rsqrt` (f16-64)
  - Truncate and round: `OPCODE` ← `roundscale` (f16-64)

```
1 __m512d a, r1, r2;
2 a = _mm512_set1_epi32(-1.75,-1.25,1.25,1.75,/*...*/); // a = [-1.75,-1.25,1.25,1.75,...]_8^f64
3 // __m512d _mm512_roundscale_pd (__m512d a, int imm8);
4 r1 = _mm512_roundscale_pd(a, _MM_FROUND_TO_ZERO); // trunc
5 // a  = [-1.75,-1.25,1.25,1.75,...]_8^f64
6 // r1 = [-1.00,-1.00,1.00,1.00,...]_8^f64
7 r2 = _mm512_roundscale_pd(a, _MM_FROUND_TO_NEAREST_INT); // round
8 // a  = [-1.75,-1.25,1.25,1.75,...]_8^f64
9 // r2 = [-2.00,-1.00,1.00,2.00,...]_8^f64
```

➔ `mask` and `maskz` variants available

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)

Intrisics – Saturated Arithmetic

- Only for addition and subtraction on low precision integers
  - ADD/SUB SATURATED: `OPCODE` ← `adds` or `subs` (i8-16 and u8-16)

```
1  _m512i u = _mm512_set_epi8(63,/*...,*/8,7,6,5,6,5,4,3,2,1,0);
2                                 // u = [0,   1,   2,   3,   4,   5,   6,   7,   8, ...,  63]_64^u8
3  u = _mm512_adds_epu8(u, u); // u = [0,   2,   4,   6,   8,  10,  12,  14,  16, ..., 126]_64^u8
4  u = _mm512_adds_epu8(u, u); // u = [0,   4,   8,  12,  16,  20,  24,  28,  32, ..., 252]_64^u8
5  u = _mm512_adds_epu8(u, u); // u = [0,   8,  16,  24,  32,  40,  48,  56,  64, ..., 255]_64^u8
6  u = _mm512_adds_epu8(u, u); // u = [0,  16,  32,  48,  64,  80,  96, 112, 128, ..., 255]_64^u8
7  u = _mm512_adds_epu8(u, u); // u = [0,  32,  64,  96, 128, 160, 192, 224, 255, ..., 255]_64^u8
8  u = _mm512_adds_epu8(u, u); // u = [0,  64, 128, 192, 255, 255, 255, 255, 255, ..., 255]_64^u8
9  u = _mm512_adds_epu8(u, u); // u = [0, 128, 255, 255, 255, 255, 255, 255, 255, ..., 255]_64^u8
10 u = _mm512_adds_epu8(u, u); // u = [0, 255, 255, 255, 255, 255, 255, 255, 255, ..., 255]_64^u8
```

  - ➔ Saturated instructions does not exist in scalar instructions
  - ➔ `mask` and `maskz` variants available

Intrinsics – Conversion (1)

- Conversion from `float` to `int` and `uint`, **round to nearest mode**
  - OPCODE ← `cvt`

```
1 __m512i _mm512_cvtps_epi32 (__m512  a); // f32_16 -> i32_16
2 __m512i _mm512_cvtps_epi64 (__m256  a); // f32_08 -> i64_08
3 __m512i _mm512_cvtps_epu32 (__m512  a); // f32_16 -> u32_16
4 __m512i _mm512_cvtps_epu64 (__m256  a); // f32_08 -> u32_08
5
6 __m256i _mm512_cvtpd_epi32 (__m512d a); // f64_8 -> i32_8
7 __m512i _mm512_cvtpd_epi64 (__m512d a); // f64_8 -> i64_8
8 __m256i _mm512_cvtpd_epu32 (__m512d a); // f64_8 -> u32_8
9 __m512i _mm512_cvtpd_epu64 (__m512d a); // f64_8 -> u64_8
```

- Conversion to/from `float` from/to `double`
  - OPCODE ← `cvt`

```
1 __m512d _mm512_cvtps_pd (__m256  a); // f32_8 -> f64_8
2 __m256  _mm512_cvtpd_ps (__m512d a); // f64_8 -> f32_8
```

➔ `mask` and `maskz` variants are available

Intrinsics – Conversion (2)

- Conversion from `float` to `int` and `uint`, with control of the **rounding mode**
  - Rounding modes are defined by IEEE 754
  - `OPCODE` ← `cvt_round`

```
1 __m512i r, __m512 a;
2 a = _mm512_setr_ps(-1.75f, -1.25f, 1.25f, 1.75f, /*...*/);  // r = [-1.75, -1.25, 1.25, 1.75, ...]_16^f32
3 // __m512i _mm512_cvt_roundps_epi32 (__m512 a, int rounding)
4 r = _mm512_cvt_roundps_epi32(a, _MM_FROUND_TO_NEAREST_INT); // r = [-2   , -1   , 1   , 2   , ...]_16^i32
5 r = _mm512_cvt_roundps_epi32(a, _MM_FROUND_TO_NEG_INF    ); // r = [-1   , -1   , 2   , 2   , ...]_16^i32
6 r = _mm512_cvt_roundps_epi32(a, _MM_FROUND_TO_POS_INF    ); // r = [-2   , -2   , 1   , 1   , ...]_16^i32
7 r = _mm512_cvt_roundps_epi32(a, _MM_FROUND_TO_ZERO       ); // r = [-1   , -1   , 1   , 1   , ...]_16^i32
```

➔ `_MM_FROUND_TO_ZERO` behaves like a truncation!

➔ `mask` and `maskz` variants are available

Intrinsics – Conversion (3)

- Conversion from `int` and `uint` to `float`
  - `OPCODE` ← `cvt`

```
1 __m512  _mm512_cvtepi32_ps (__m512i a); // i32_16 -> f32_16
2 __m512d _mm512_cvtepi32_pd (__m256i a); // i32_16 -> f64_16
3 __m512  _mm512_cvtepu32_ps (__m512i a); // u32_16 -> f32_16
4 __m512d _mm512_cvtepu32_pd (__m256i a); // u32_16 -> f64_16
5
6 __m256  _mm512_cvtepi64_ps (__m512i a); // i64_8 -> f32_8
7 __m512d _mm512_cvtepi64_pd (__m512i a); // i64_8 -> f64_8
8 __m256  _mm512_cvtepu64_ps (__m512i a); // u64_8 -> f32_8
9 __m512d _mm512_cvtepu64_pd (__m512i a); // u64_8 -> f64_8
```

➔ `mask` and `maskz` variants are available

Intrinsics – Ordering instructions (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

Intrinsics – Ordering instructions (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, `OPCODE` ← `min` or `max`

```
1 for (int i = 0; i < n; i += 64) {
2   __m512i a = _mm512_loadu_si512(A + i);
3   __m512i b = _mm512_loadu_si512(&B[i]);
4   __m512i m = _mm512_min_epu8(a, b);
5   __m512i M = _mm512_max_epu8(a, b);
6   _mm512_storeu_si512(Min + i, m);
7   _mm512_storeu_si512(&Max[i], M);
8 }
```

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

➔ `mask` and `maskz` variants available

Intrinsics – Comparison instructions

- Comparison of two vector registers generates a **mask register**
  - OPCODE ← cmp

```
1  __mmask8 k1, k2, k3, k4, k5, k6;
2  __m512d a = _mm512_set_pd(0,1,2,3,4,5,6,7);            // a = [7,6,5,4,3,2,1,0]_8^f64
3  __m512d b = _mm512_setr_pd(1,2,3,4,5,6,7,8);           // b = [1,2,3,4,5,6,7,8]_8^f64
4  // __mmask8 _mm512_cmp_pd_mask (__m512d a, __m512d b, const int imm8);
5  k1 = _mm512_cmp_pd_mask(a, b, _CMP_LT_OS);  // a <  b -- k1 = [0,0,0,0,1,1,1,1]_8^b
6  k2 = _mm512_cmp_pd_mask(a, b, _CMP_LE_OS);  // a <= b -- k2 = [0,0,0,1,1,1,1,1]_8^b
7  k3 = _mm512_cmp_pd_mask(a, b, _CMP_EQ_OS);  // a == b -- k3 = [0,0,0,1,0,0,0,0]_8^b
8  k4 = _mm512_cmp_pd_mask(a, b, _CMP_NEQ_OS); // a != b -- k4 = [1,1,1,0,1,1,1,1]_8^b
9  k5 = _mm512_cmp_pd_mask(a, b, _CMP_GE_OS);  // a >= b -- k5 = [1,1,1,1,0,0,0,0]_8^b
10 k6 = _mm512_cmp_pd_mask(a, b, _CMP_GT_OS);  // a >  b -- k6 = [1,1,1,0,0,0,0,0]_8^b
```

  - Exist for all the datatypes (f16-64, i8-64 and u8-64)
  - ➔ `mask` and `maskz` variants available

Intrinsics – Design Pattern Selection (1)

- 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(c,b));`
  - Assumption comparisons return `0x00` or `0xff`
- In SIMD
  - `c=vclt_u8(a,b); m=vorr_u8(vand_u8(a,c),vbic_u8(b,c)));`
- AVX-512 dedicated instruction for the selection:
  - `OPCODE` ← `blend`
  - Almost all instructions have an integrated `blend` ➔ `mask` and `maskz` variants

Intrinsics – Design Pattern Selection (2)

- OPCODE ← blend

```
1 __m512i _mm512_mask_blend_epi8  (__mmask64 k, __m512i a, __m512i b);
2 __m512i _mm512_mask_blend_epi16 (__mmask32 k, __m512i a, __m512i b);
3 __m512i _mm512_mask_blend_epi32 (__mmask16 k, __m512i a, __m512i b);
4 __m512i _mm512_mask_blend_epi64 (__mmask8  k, __m512i a, __m512i b);
5
6 __m512h _mm512_mask_blend_ph    (__mmask32 k, __m512h a, __m512h b);
7 __m512  _mm512_mask_blend_ps    (__mmask16 k, __m512  a, __m512  b);
8 __m512d _mm512_mask_blend_pd    (__mmask8  k, __m512d a, __m512d b);
```

- Performs: `r_i = k_i ? a_i : b_i;`
- No other variant
- Keep in mind that `blend` is generally useless in AVX-512

Web References

4 AVX-512 SIMD Programming

- Intel Intrinsics Reference (**your best friend**)
  `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`
- AVX-512: Architecture and ISA
  - Skylake `https://colfaxresearch.com/skl-avx512/`
  - Skylake `https://www.microway.com/knowledge-center-articles/detailed-specifications-of-the-skylake-sp-intel-xeon-processor-scalable-family-cpus/`
  - ISA `https://en.wikipedia.org/wiki/AVX-512`
  - EVEX `https://en.wikipedia.org/wiki/EVEX_prefix`
  - VNNI `https://en.wikichip.org/wiki/x86/avx512_vnni`
- SIMD wrappers
  - MIPP `https://github.com/aff3ct/MIPP`
  - Highway `https://github.com/google/highway`
  - nSIMD `https://github.com/agenium-scale/nsimd`
  - xSIMD `https://github.com/xtensor-stack/xsimd`

Q&A

Thank you for listening!
Do you have any questions?