

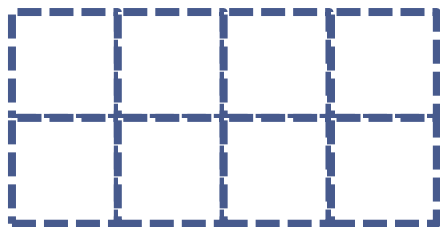
SIMD

- ▶ SIMD = Single Instruction Multiple Data
- ▶ Hardware support
 - ▶ Vector registers
 - wide registers (64-512 bits), interpretation depends on instructions used
 - in some architectures, (lower) parts of wide registers act as smaller vector (or even scalar) registers (backward compatibility)
 - ▶ Vector instructions
 - act on vector registers similarly to normal instructions acting on scalar registers
 - what humans call vectors, hardware sees as scalars
 - logically, each vector instruction performs N mathematical operations at once
 - In most cases, the N **lanes** act independently
 - physically, the N operations may be executed:
 - all in the same moment, using N hardware units
 - in a pipeline, using single hardware unit (usually divided into stages)
 - combined, feeding N/K batches into K hardware units
 - scalability: different hardware may use different K for the same instruction
 - different instructions use different vector elements (double, float, int64,...,int8)
 - instructions have different N (therefore K)
 - the same hardware (e.g. an adder) is reconfigured into different K's (e.g. by cutting carry)

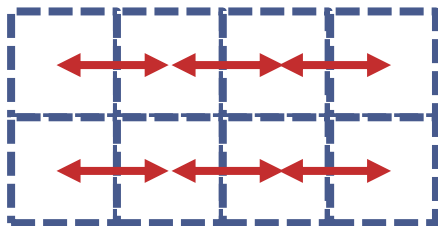
Different processing strategies

time
↓

$N=8, K=4$



Example: Older implementations of AVX

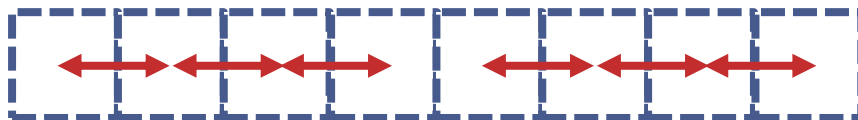


Problem: Cross-lane instructions (e.g. shuffles) cannot pass data between the two halves

$N=8, K=8$



Example: Newer implementations of AVX using either a 256-bits wide pipeline or two 128-bit pipelines synchronously

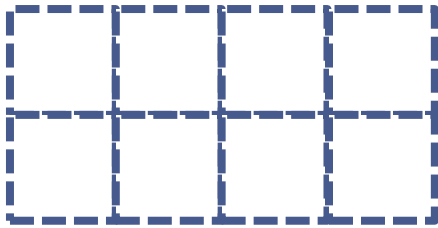


The problem remains visible in the AVX instruction set

Different lane width

time
↓

N=8, K=4

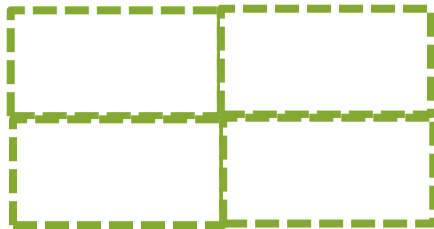


N=8, K=8



Example: Single-precision FP in AVX
 $8 * 32 = 256$ bits

N=4, K=2



N=4, K=4



Example: Double-precision FP in AVX
 $4 * 64 = 256$ bits

► Hardware support

- Vector registers
- Vector instructions
- Memory transfers
 - in most cases, vector registers must be read/written from/to contiguous blocks of memory
 - existence of vector instructions requires widening of internal data paths in CPU
 - similarly to arithmetics, one vector may be transferred either at once or as a series of smaller batches (since ultra-wide data paths are expensive)
 - the batches (usually) originate in the same cache line – only one cache lookup needed
 - even if the data paths were not wider than a scalar, vector transfers would be faster
 - there are soft or hard requirements for alignment
 - Align to data path width (16/32 bytes in current Intel/AMD CPUs)
 - Do not cross cache line boundaries (64 bytes in all current Intel/AMD CPUs)

► Software support

- Automatic vectorization by compilers
 - The transformation is often non-equivalent wrt. strict language rules
 - Explicit permission from the programmer is needed (pragma)
 - Advanced transformation methods now called “polyhedral compilation” (e.g. Polly/LLVM)
- Vectorized library code
 - Operations on arrays/matrices implemented using vector instructions
- Explicit use of vector datatypes and instructions
 - Make use of all instructions available, including peculiarities
 - In assembly languages – error prone and often worse than product of compilers
 - In higher languages – using *intrinsic functions*
 - Compilers take care of register allocation, addressing, type safety, etc.
- Handling alignment requirements
 - All parts (programmers, compilers, libraries) must cooperate to make data properly aligned

- ▶ SIMD support in Intel/AMD CPUs
 - ▶ MMX (Intel 1997)
 - 64 bits, 8 registers (MM0..7), shared with scalar floating-point unit (x87)
 - only integer operations (8/16/32-bit), targeted at audio processing
 - AMD 3DNow added some 32-bit floating point support
 - ▶ SSE (Intel 1999)
 - 128 bits, 8 registers (XMM0..7), only 32-bit floating point supported
 - ▶ SSE2-SSE4 (Intel 2001-2007)
 - 64-bit floating-point and 8/16/32/64-bit integer arithmetics for 128-bit vectors
 - ▶ x64 (AMD 2003)
 - additional 8 registers (XMM8..15) available in 64-bit mode
 - ▶ AVX (Intel/AMD 2011)
 - 256 bits, 16 registers (YMM0..15) (only YMM0..7 accessible in 32-bit mode)
 - floating point (32/64-bit) operations only
 - three-operand instruction format
 - ▶ AVX2 (Intel 2013)
 - integer arithmetics (8/16/32/64-bits) extended to 256-bit vectors
 - gather/maskstore instructions

- ▶ SIMD support in Intel/AMD CPUs
 - ▶ AVX (Intel/AMD 2011)
 - 256 bits, 16 registers (YMM0..15) (only YMM0..7 accessible in 32-bit mode)
 - floating point (32/64-bit) operations only
 - three-operand instruction format
 - ▶ AVX2 (Intel 2013)
 - integer arithmetics (8/16/32/64-bits) extended to 256-bit vectors
 - gather/maskstore instructions
 - ▶ IMCI (Intel 2012)
 - in Intel Knights Corner architecture (aka. MIC aka. Xeon Phi)
 - 512 bits, 32 registers (ZMM0..31)
 - gather/scatter instructions
 - mask registers
 - ▶ AVX512 (Intel 2016)
 - in Intel Knights Landing (aka. MIC 2 aka. Xeon Phi second generation)
 - in Intel Skylake Purley (2017), Cannonlake (2018)
 - most instructions equivalent to IMCI (but different binary encoding)

► Advantages of SIMD

► Greater arithmetic throughput

- double-precision multiply on Skylake: 2×4 operations per clock cycle (vs. 2 scalar)
 - fused multiply-add (FMA): 2×4 muls + 2×4 adds per clock
- 32-bit integer addition on Skylake: 3×8 operations per clock (vs. 4 scalar)

► Greater memory throughput

- Only vector instructions can use the full 256-bit width of CPU-L1 bus
- Vector throughput: 64B loads + 32B stores per clock
 - Scalar double-precision throughput: 16B loads + 8B stores

► Greater register file

- scalar x64 integer: $16 \times 64\text{bit} = 128$ bytes
- scalar extended-double-precision: $8 \times 80\text{bit} = 80$ bytes
- AVX2: $16 \times 256\text{bit} = 512$ bytes
- AVX512: $32 \times 512\text{bit} = 2048$ bytes
 - for comparison: Xeon Phi L1 Cache = 64 KB shared by 4 threads = 16 KB per thread

Vector data types and registers (MMX/SSE/AVX/AVX512)

A
HL
BC
DE



SP



FLAGS



PC

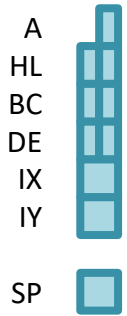


1974

Intel 8080

12 B of registers

64 KB addressable memory



1976

Zilog Z80

16 B of (app) registers

64 KB addressable memory

AX
BX
CX
DX
SI
DI
BP
SP



FLAGS



IP



1978

Intel 8086

20 B of (app) registers

1 MB addressable memory

AX
BX
CX
DX
SI
DI
BP
SP



FLAGS
IP



1980
Intel 8086+8087
100 B of (app) registers
1 MB addressable memory

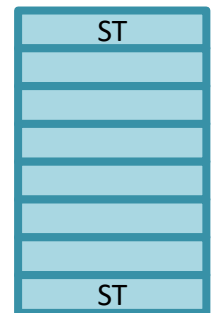
*8 80-bit FP
registers in
co-processor
chip (8087)*



*this picture is shown
with MSB on the left*



1985
Intel 80386+80387
120 B of app registers
4 GB addressable memory



*this picture is shown
with MSB on the left*



EFLAGS

EIP

1989
Intel 80486
120 B of app registers
4 GB addressable memory

*The FPU is
now on the
same chip*

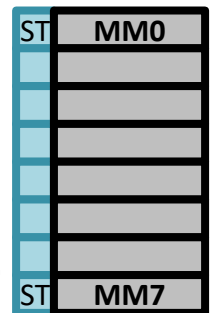


*this picture is shown
with MSB on the left
(lane 0 on the right)*



1997
Intel Pentium MMX
120 B of app registers
4 GB addressable memory

*64-bit MMX
registers were
carved from
the 80-bit FP
registers (x87)*



Scalar and vector registers (MMX/SSE) – 32 bit mode

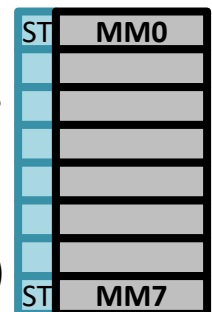


*this picture is shown
with MSB on the left
(lane 0 on the right)*

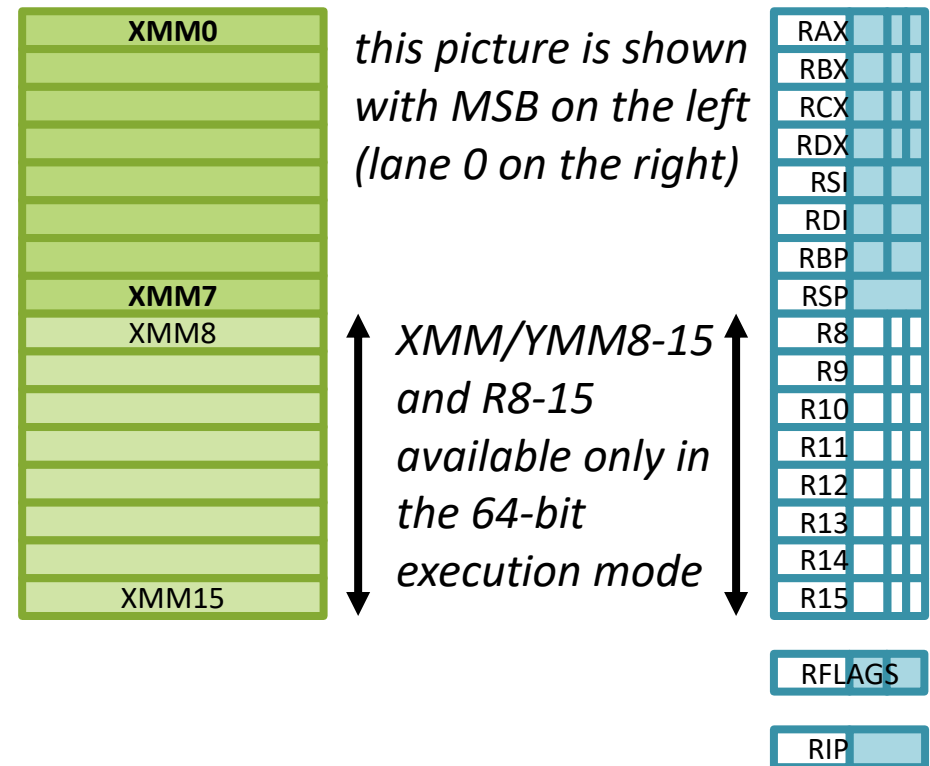


1999
Intel Pentium III
248 B of app registers
4 GB addressable memory

*64-bit MMX
registers were
carved from
the 80-bit FP
registers (x87)*

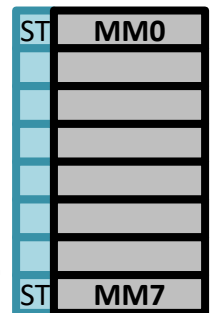


Scalar and vector registers (MMX/SSE) – 64 bit mode



2003
AMD Opteron
480 B of app registers
1 TB addressable memory

64-bit MMX registers were carved from the 80-bit FP registers (x87)



Scalar and vector registers (IMCI)

2010

**Intel Xeon Phi
Knights Corner
2272 B of app
registers
per thread**

ZMM0

ZMM31

*this picture is shown
with MSB on the left
(lane 0 on the right)*

*The first
Knights Corner
CPUs were
derived from a
Pentium core
converted to 64
bits and had no
support for SSE
or MMX*

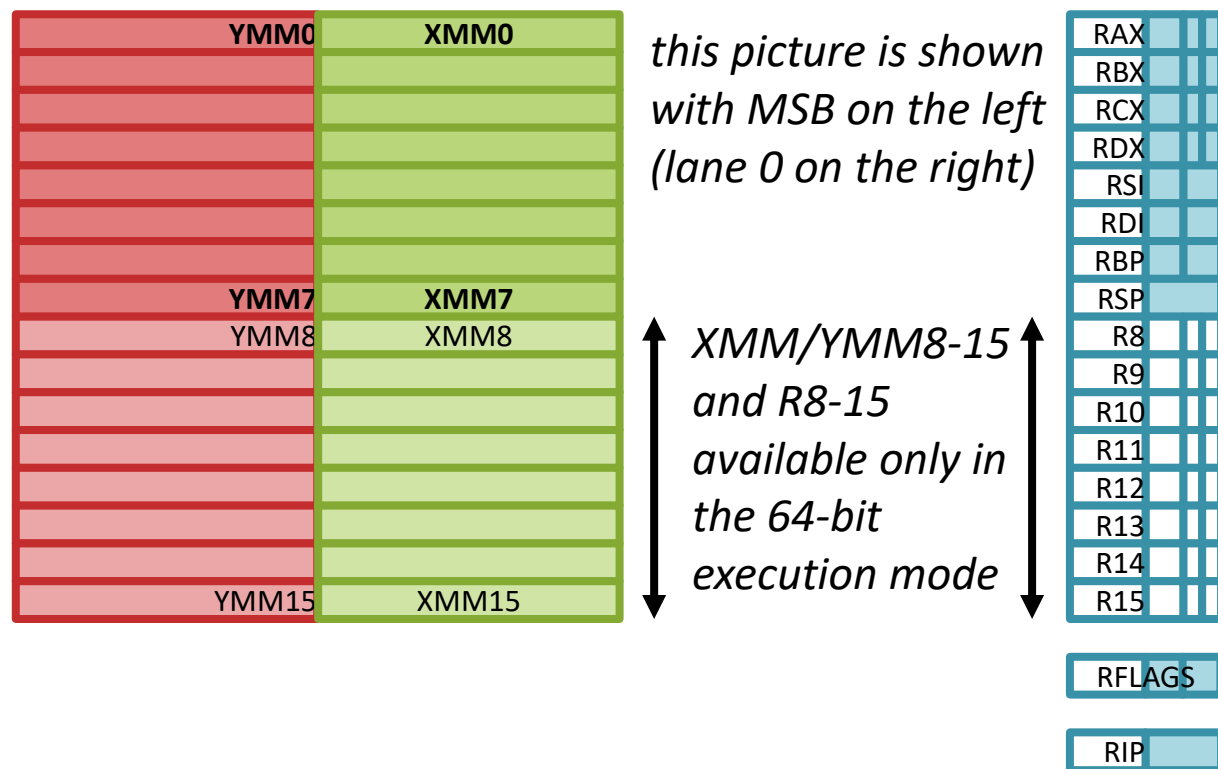
RAX			
RBX			
RCX			
RDX			
RSI			
RDI			
RBP			
RSP			
R8			
R9			
R10			
R11			
R12			
R13			
R14			
R15			

RFLAGS

RIP

ST
ST

Scalar and vector registers (MMX/SSE/AVX)

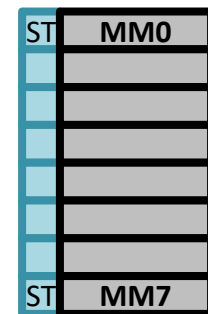


2011

Intel Sandy Bridge

736 B of app registers per thread

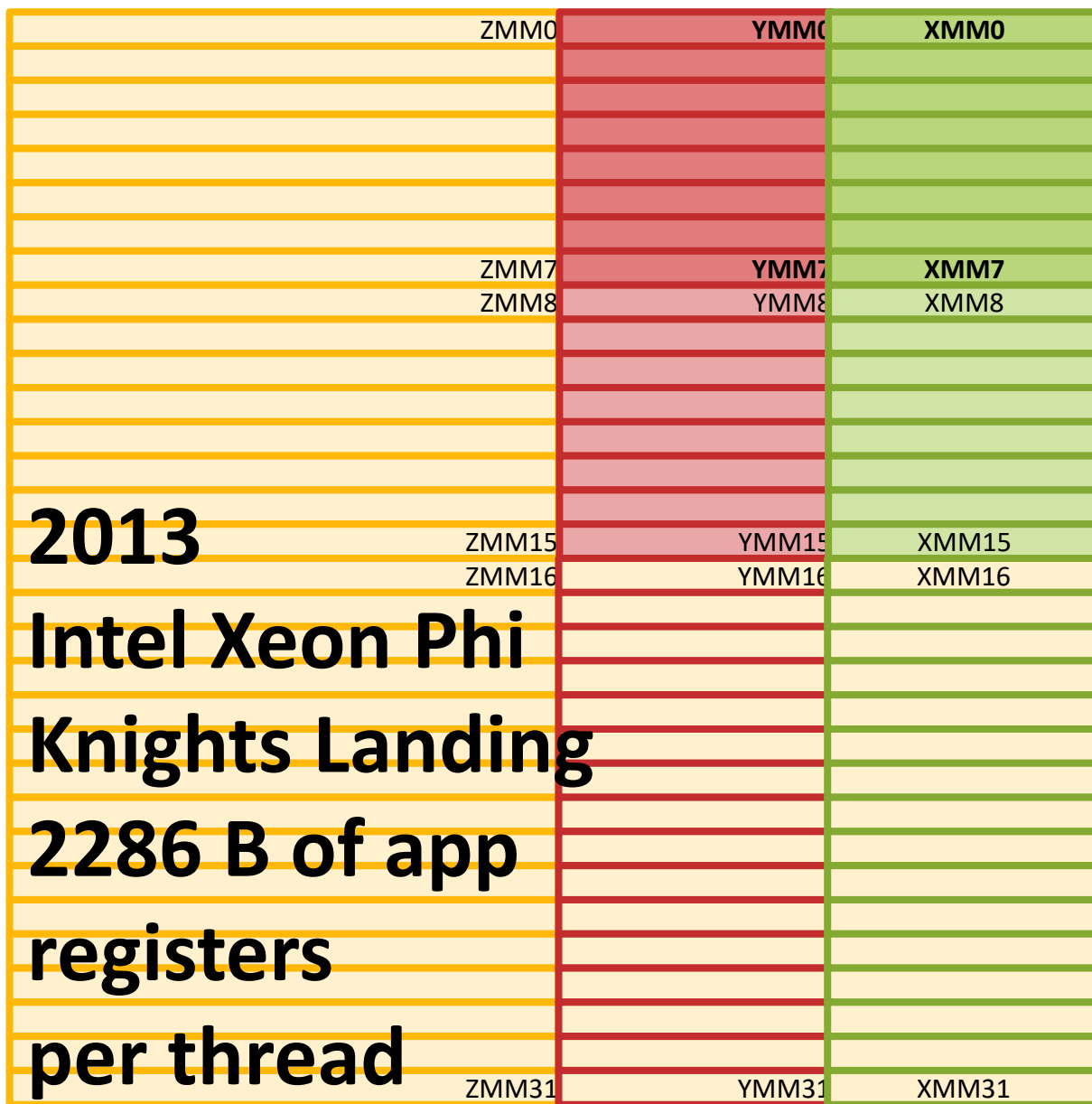
1 TB addressable memory



Scalar and vector registers (MMX/SSE/AVX/AVX512)

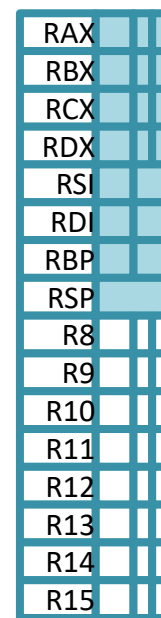
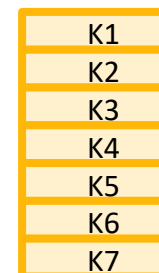
2013 Intel Xeon Phi Knights Landing 2286 B of app registers per thread

ZMM15
ZMM16
ZMM31



*this picture is shown
with MSB on the left
(lane 0 on the right)*

AVX512 mask registers



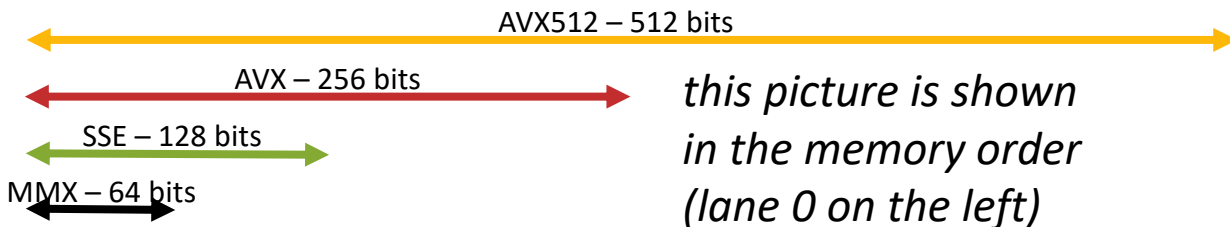
RFLAGS

RIP



Vector data types (MMX/SSE/AVX/AVX512)

*C intrinsic
function suffix*



*this picture is shown
in the memory order
(lane 0 on the left)*

*assembler
instruction
suffix (Intel)*

pd



PD

ps



PS

epi128/si128



only bitwise and/or/xor and shift instructions

DQ

or none

epi64 (signed)

epu64 (unsigned)



Q

epi32 (signed)

epu32 (unsigned)



D

epi16 (signed)

epu16 (unsigned)



w

epi8 (signed)

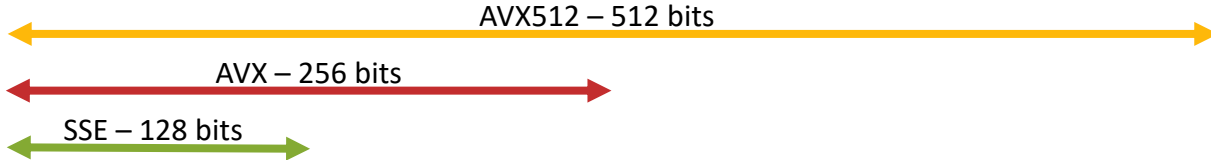
epu8 (unsigned)



B

Floating-point vector data types (MMX/SSE/AVX/AVX512)

*C intrinsic
function suffix*



*assembler
instruction
suffix (Intel)*

pd



PD

ps



PS

AVX-512_BF16 - since Cooper Lake (Intel 2020):

pbh



BF16

AVX-512_FP16 - since Sapphire Rapids (Intel 2023):

ph



PH

Bits shown with MSB on the left; opposite of the memory order:

pd = IEEE 754 Double



ps = IEEE 754 Single



pbh



ph = IEEE 754 Half



Vector instructions (SSE/AVX/AVX512)

Vector instructions (SSE/AVX/AVX512)

- ▶ **Memory access**
 - ▶ Loads and stores
 - ▶ In addition, many other instructions may have up to 1 memory operand
- ▶ **Plain arithmetic instructions**
 - ▶ Parallel execution of the same operation in each lane, independently
- ▶ **Conditions and masks**
 - ▶ Support for conditional execution, independently in each lane
- ▶ **Inter-lane arithmetics**
 - ▶ Applying selected operations across lanes
- ▶ **Inter-lane shuffles**
 - ▶ Movement of data between lanes
- ▶ **Conversions**
 - ▶ Changing widths of data; interaction with scalar registers
- ▶ The list in these slides can never be complete, see the reference:

<https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>

Vector instructions (SSE/AVX/AVX512)

▶ Memory access

▶ Vector load and stores

- Load/store vector from/to a consecutive block of memory
- **load/store** - Aligned loads/stores – fault if unaligned to a multiple of 16B
- **loadu/storeu** - Unaligned load/stores – slower if unaligned
 - In older architectures, slower even if aligned – use only if alignment cannot be guaranteed
- **stream_load** - Non-temporal loads – not stored in caches (where architecture allows)
 - To avoid cache pollution when the data will not be read again soon

▶ Memory arguments of vector instructions

- At most one argument may reside in memory
- In SSE, the memory argument must always be aligned to 16B
- In AVX-enabled CPUs, memory arguments may be unaligned (resulting in slower operation)
 - Applies also to SSE instructions, if VEX-encoded (assembler names prefixed by 'V')

▶ When used from C/C++

- The compiler automatically generates loadu/storeu (or memory arguments if AVX is enabled) whenever working with memory operands
- If alignment is guaranteed, explicit load/store to a local variable usually produces faster code

► Memory access and data types

- Formally, a vector load/store of a particular width (128/256/512 bits) just moves the bits between memory and a (XMM/YMM/ZMM) register, independently of the lane size and format
- Physically, floating-point vectors may be routed through different parts of the CPU than integer vectors
 - And, in theory, different lane widths may also have different pathways
 - This arrangement makes the data closer to the respective hardware units for the following/preceding arithmetic instructions
- Therefore, there are (at least) three kinds of instructions for loads/stores
 - And even more intrinsic functions mapped to them
 - `*MOVDQ(A|U) = *(load|store)[u]_e[si(32|64|128)] = integer loads/stores`
 - `*MOV(A|U)PS = *(load|store)[u]_ps = float loads/stores`
 - `*MOV(A|U)PD = *(load|store)[u]_pd = double loads/stores`
- Always use the form of load/store related to the arithmetic instructions which operate on the data
 - With intrinsic functions in C/C++, this is enforced by the existence of different data types representing integer/float/double vectors
 - In assembly language, there is no enforcement

Vector instructions (SSE/AVX/AVX512)

► Memory access

► Gather/scatter

- Gather available from AVX2, scatter only for AVX512 (and KNC)
- Load/store lanes of a vector from/to individually indexed positions
 - Available only for 32 or 64 bit data elements
 - Addresses computed by adding a common base address and (the 1/2/4/8-multiple of) an 32/64 bit index in the corresponding lane of a vector register
 - Index and data register may differ in size (e.g., `_mm256_i32gather_epi64` reads indexes from a 128-bit register and stores to a 256-bit)
- The CPU may perform individual lane loads/stores in parallel
 - if they do not hit the same parts of the internal memory buses
 - similar to the notion of *stride* in GPUs but far less massive
- In any case, gather/scatter is slower than contiguous loads/stores
 - But faster than a series of scalar load/stores
- Gather:

```
for (i in 0..N-1) v[i] = a[c*x[i]]
```

- `c` is a constant of 1/2/4/8

- For scatter, individual lanes may be masked by a bit-mask:

```
for (i in 0..N-1) if (m[i]) a[c*x[i]] = v[i]
```

- Beware: If two indexes are identical in the same scatter, the result is undefined
 - Use **conflict** instructions (in AVX512CD) to detect, then masking to avoid

► Plain arithmetic instructions

- Parallel execution of the same operation in each lane, independently

```
for (i in 0..N-1) c[i] = f(a[i],b[i])
```

- Integer arithmetics:

- ADD, SUB in 8/16/32/64 bit lanes
- Saturated signed/unsigned ADD/SUB in 8/16 bit lanes
- Shifts in 16/32/64/128 bit lanes
- MUL in 32/64 bit lanes

- Floating-point arithmetics (32/64-bit lanes)

- ADD, SUB, MUL, DIV
- FMA (fused multiply-add)

```
d[i] = c[i] + a[i]*b[i]
```

- DP (dot product) with extension pbh->ps

```
d[i] = c[i] + a[2*i]*b[2*i] + a[2*i+1]*b[2*i+1]
```

Vector instructions (SSE/AVX/AVX512)

► Conditions and masks

- Support for conditional execution, independently in each lane

► SSE and AVX

- Comparisons produce all-ones (-1) or all-zeros (0) in each lane
 - Only EQ and GT supported for integers, the others must be derived
 - All six comparisons supported for floating point
- Conditional expressions are simulated using bitwise AND, ANDNOT, and OR:
 - BEWARE: ANDNOT negates the FIRST argument before anding

```
// for (i in 0..N-1) e[i] = a[i] == b[i] ? c[i] : d[i]
```

```
cond = cmpeq(a,b)           // cond[i] = a[i] == b[i] ? -1 : 0
```

```
left = and(cond,c)          // left = cond & c
```

```
right = andnot(cond,d)      // right = ~cond & d
```

```
e = or(left,right)          // e = left | right
```

- The three bitwise operators come in three flavors, depending on type
 - The reason is the same as for loads/stores
 - `*P(AND|ANDN|OR) = *(and|andnot|or)_si128`
 - `*(AND|ANDN|OR)PS = *(and|andnot|or)_ps`
 - `*(AND|ANDN|OR)PD = *(and|andnot|or)_pd`

Vector instructions (SSE/AVX/AVX512)

► Conditions and masks

- Support for conditional execution, independently in each lane

► AVX512

- 7 special mask (K) registers containing single bit for each lane
 - The number of lanes used depend on the instruction
 - Presented as types `__mmask[8|16|32|64]` in C/C++
- Comparisons produce one bit for each lane
 - All six comparisons supported for all types
- Almost all instructions have masked variants
 - The instruction is applied for the lanes which have 1 in the corresponding mask operand lane
 - In the other lanes, the result register retains the previous value
 - Comparison instructions may be masked too – used to simulate Boolean conjunction
- In C/C++ intrinsics, masking is presented in two forms
 - **mask** – two additional inputs: previous value vector `src` and mask vector `k`:

```
for (i in 0..N-1) r[i] = k[i] ? f(a[i],b[i]) : src[i]
```

- **maskz** – one additional input: mask vector `k`, masked lanes produce zero:

```
for (i in 0..N-1) r[i] = k[i] ? f(a[i],b[i]) : 0
```

- Conditional expressions and statements are simulated using masking:
 - The same mechanism is used in GPUs

```
// for (i in 0..N-1) e[i] = a[i] == b[i] ? c[i] : d[i]
```

```
cond = cmpeq(a,b)           // cond is a mask register
```

```
e = mask_mov(right,cond,left) // e[i] = cond[i] ? left : right
```

Vector instructions (SSE/AVX/AVX512)

► Inter-lane arithmetics

- Applying selected operations across lanes
- **hadd/hsub** - Horizontal ADD/SUB (16/32/float/double lanes)
 - SSE version:

```
for(i in 0..N/2-1) {  
    r[i] = f(a[2*i],a[2*i+1])  
    r[N/2 + i] = f(b[2*i],b[2*i+1])  
}
```

- AVX version acts as applying SSE version to each half of the vectors
 - A consequence of implementing AVX using 128-bit pipelines

```
for(i in 0..N/4-1) {  
    r[i] = f(a[2*i],a[2*i+1])  
    r[N/4 + i] = f(b[2*i],b[2*i+1])  
    r[N/2 + i] = f(a[N/2 + 2*i],a[N/2 + 2*i+1])  
    r[N*3/4 + i] = f(b[N/2 + 2*i],b[N/2 + 2*i+1])  
}
```

- Effectively swaps the middle two quarters wrt. the naturally expected behavior
- There is no AVX512 version, operands must be first split into pairs of AVX vectors

Vector instructions (SSE/AVX/AVX512)

► Inter-lane shuffles

- Movement of data between lanes
- BEWARE: Most AVX/AVX512 shuffle instructions cannot move data between the 128-bit halves/quarters of the vectors
 - Consequence of the original implementation using 128-bit pipelines
 - Use **permute2f128** for movement across AVX halves, **permute4f128** for AVX2
 - A vector-wide shuffle must be combined from permute and a 128-wide shuffle
- ***alignr_epi8** – byte-granular shift right
 - concatenate two 128-bit vectors, then pick 128 bits at the specified location
 - the shift amount must be a constant (embedded into the instruction)
- AVX512: ***alignr_epi(32|64)** – 4/8-byte-granular shift right
 - works smoothly across 128-bit boundaries
- ***permute***, ***shuffle*** – “arbitrary” permutations
 - unary cases

```
for (i in 0..N-1) r[i] = a[p[i]]
```

- binary cases

```
for (i in 0..N-1) r[i] = (p[i]&TOP_BIT) ? b[p[i]&LOW_BITS] : a[p[i]&LOW_BITS]
```

- many variants differing in granularity and other limitations
- most variants require permutation encoded in a constant, few accept run-time values

► Conversions

- Changing widths of data; interaction with scalar registers
- ***extract*** - copy selected lane into a scalar register (or smaller vector register)
 - the lane index must be a constant
- ***insert*** - copy a scalar value into a selected lane of a vector register
 - the rest remains untouched, therefore there is an input vector too
 - the lane index must be a constant
- ***broadcast*** - copy a scalar value (or a smaller vector) into all lanes

► Pseudo-intrinsic functions (not single instructions) in C/C++

- ***set1*** - same as broadcast (where not in instruction set)
- ***setzero*** - set all lanes to zero
- ***cast*** - conversion between various vector forms (no runtime operation)

Using MMX/SSE/AVX intrinsics in C/C++

▶ Intrinsic functions

- ▶ Formally declared in header files
- ▶ Recognized by the compiler
 - Most intrinsic functions expand to one vector instruction
 - Some functions are implemented using more than one scalar or vector instruction
- ▶ De-facto standard dictated by Intel and copied by MSVC, gcc, and others

▶ Data types

- ▶ Declared in header files together with functions
- ▶ Names are standardized, but contents is different (use only as black boxes)
- ▶ Data types correspond to vector register types (widths)

`__m64`, `__m128`, `__m256`, `__m512`

- For some type safety, there are three types for each width
 - single-precision (no suffix)
 - double-precision (suffix 'd')
 - half-precision (suffix 'bh' for BF16 or 'h' for IEEE 754 Half)
 - all integer widths (suffix 'i', except of `__m64`)

Using MMX/SSE/AVX intrinsics in C/C++

header file	types	functions	technology
mmintrin.h	__m64		MMX
xmmintrin.h	__m128	_mm*_ps	SSE
emmintrin.h	__m128d, __m128i	_mm*_pd _mm*_ep(i u)(8 16 32 64)	SSE2
pmmmintrin.h		_mm*_p(s d)	SSE3
tmmintrin.h		_mm*_epi(8 16 32)	SSSE3
smmintrin.h		_mm*_*	SSE4.1
nmmintrin.h		_mm_cmp*, _mm_crc32_*, _mm_popcnt_u(32 64)	SSE4.2
wmmmintrin.h		_mm_aes*_si128	
immintrin.h	__m256, __m256d, __m256i	_mm256_*	AVX, AVX2
	__m512, __m512d, __m512i	_mm512_*	AVX512
ammintrin.h		_mm_*, _mm256_*	AMD extensions

Using MMX/SSE/AVX intrinsics in C/C++

► Alignment

- It is recommended to align all vectors to 16 bytes. If not 16-byte aligned:
 - SSE-only CPUs: **segfault** except for MOVUPS (loadu/storeu)
 - AVX-enabled CPUs: **reduced throughput**, no segfault except for MOVAPS (load/store)
- It is advisable to align AVX2 vectors to 32 bytes and AVX512 vectors to 64 bytes
 - avoid splitting over cache-line boundary (a split load counts as two loads)
- Compiler support
 - When vector types are used for static or local variables or their parts, the compiler will align them
 - Some compilers align to 16 and use MOVUPS, others align AVX to 32 and use MOVAPS

```
__m256i v1; __m256i v2[4]; std::array<__m256i,4> v3; // everything aligned
```

- When vectors are simulated as arrays of scalar types, variables are unaligned

```
std::int32_t v1[8]; // aligned only to 4 bytes!!!
```

- alignment may be enforced by **alignas(16)**

► Library support

- C++ library (containers, smart pointers) align correctly **only since C++17**

```
std::vector<__m256i> v4; // aligned only to 8 bytes before C++17
```

- Before C++17 (or in C), alignment is done via semi-standardized functions

```
_mm_malloc, posix_memalign, std::align
```

► Alignment

► alignas specifier

- Attached to class/struct types

```
struct alignas(16) aligned_chunk { std::int32_t a[4]; };
```

- Attached to variables (including class/struct members)

```
alignas(16) std::int32_t v1[8];
```

► Notes

- Alignment on dynamic allocation cannot be enforced when allocating primitive types

```
std::vector<alignas(16) std::int32_t> // SYNTAX ERROR
```

▶ Vector-unaware data structures

```
std::int32_t v1[40];
```

```
std::vector<std::int32_t> v2;
```

- Aligned only to the scalar unit
- Requires `reinterpret_cast` for vector access

▶ Vector-aware data structures

```
__m256i v3[5];
```

```
std::vector<__m256i> v4;    // requires C++17 for proper alignment
```

- Correctly aligned data
- Requires `reinterpret_cast` for scalar access

► **beware:** reinterpret_cast may violate aliasing rules of C++

- Pointers to different types (unrelated by inheritance) are considered non-aliased
 - Except the case of T* vs. char* (required for reading/writing binary files)

```
__m256i* vector_data = /*...*/;
```

```
vector_data[0] = /*...*/;
```

```
std::int32_t* scalar_data = reinterpret_cast<std::int32_t*>(vector_data);
```

```
/*...*/ = scalar_data[0];
```

- The compiler is allowed to reorder the vector store after the scalar read

► C++23: use std::launder

```
__m256i* vector_data = /*...*/;
```

```
vector_data[0] = /*...*/;
```

```
std::int32_t* scalar_data = std::launder(reinterpret_cast<std::int32_t*>(vector_data));
```

```
/*...*/ = scalar_data[0];
```

- The call to launder acts as a fence:
 - Accesses through vector_data cannot be moved past the launder call
 - Accesses through scalar_data cannot be moved before the launder call

▶ Correcting alignment at run time

- Determine alignment using $(p \% 16)$
 - requires `reinterpret_cast` to `std::intptr_t`
- ▶ When working on one unaligned array
 - Initial and final unaligned elements processed in scalars, the rest in vectors
- ▶ When working on more unaligned arrays
 - One of the arrays (preferably the output one) dictates alignment
 - Write initial/final elements as scalars, the rest as vectors
 - The other arrays:
 - Either read/written unaligned (requires AVX-enabled CPUs)
 - Or use **`alignr`** to extract the matching arguments from a pair of aligned vectors
 - Problem: **`alignr`** requires a **constant** as the shift amount
 - Code must be replicated for every possible value of alignment (may be too many)
 - Complex templated machinery in C++ may be used
 - Problem #2: AVX version of **`alignr`** works independently on 16-byte halves
 - This is a consequence of (original) implementation using pipelined 128-bit ALU
 - Use another instruction (**`permute2f128`**) before **`alignr`**