Single Instruction Multiple Data (SIMD)

High Performance Computing - FMI, Fall 2015

martin.krastev@chaosgroup.com

Terminology

Flynn's taxonomy

	Single instruction stream	Multiple instruction stream
Single data stream	SISD	MISD
Multiple data stream	**SIMD**	MIMD

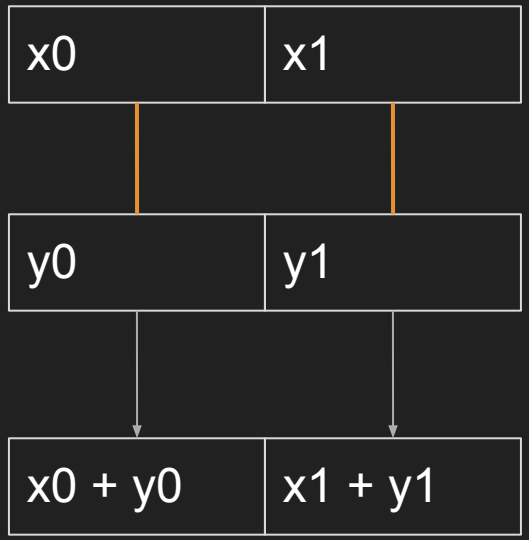
Michael Flynn, 1966

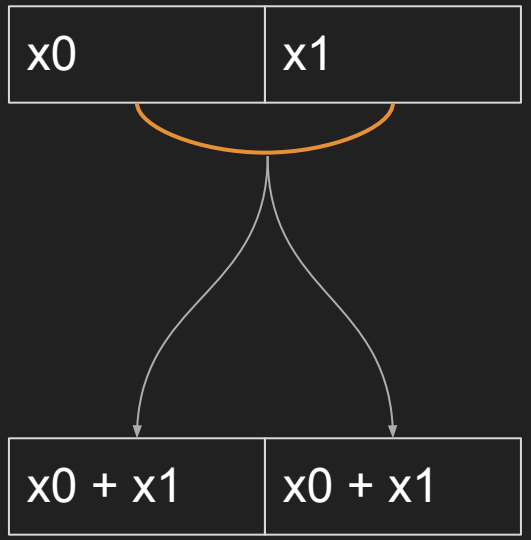
Terminology, cont'd

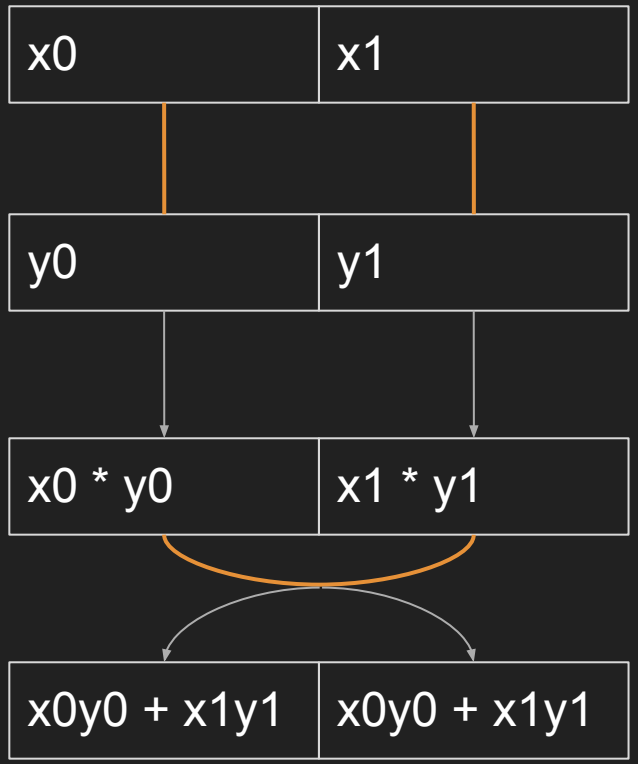
Multiplicity enables **parallelism**

Bit-Level Parallelism (BLP)	machine word width
Instruction-Level Parallelism (ILP) + Memory-Level Parallelism (MLP)	superscalarity / VLIW
Data Level Parallelism (DLP)	Vector CPU / SIMD / GPU / SPMD
Task-Level Parallelism (TLP)	Multi-tasking / multi-threading

Vertical vs horizontal ops







Vertical vs horizontal ops, cont'd

Beware the asymptotic time complexity!

- Vertical ops – O(1) via parallelism
- Horizontal ops == vector reduction
  - O(log2(N)) for associative ops, via parallelism
  - O(N) for non-associative ops, no parallelism

SIMD yields top gain for vertical ops, some gain for reduction ops using assoc. ops, and no gain for reduction ops using non-assoc. ops (ergo nobody does it).

x86 SIMD: Intel & AMD

- MMX – 1997, Pentium w/ MMX; misc. int
- SSE – 1999, Pentium 3; fp32 and int32
- SSE2 – 2001, Pentium 4; fp64, misc. int
- SSE3 – 2004, Prescott; some horizontal ops
- SSSE3 – 2006, Core ‘Merom’; integers ops
- SSE4.1 – 2007, Core ‘Penryn’; horiz. & lane
- SSE4.2 – 2008, Core ‘Nehalem’; string ops
- AVX – 2011, Core ‘Sandy Bridge’; 3-operand, 256-bit regs
- AVX2 – 2013, Core ‘Haswell’; permutes, int. ops, FMA3

x86 SIMD: AMD-only

- 3DNow! – 1998, K6-2; fp32 & cache control
- SSE4a – 2007, ‘Barcelona’; bit-level-access
- XOP – 2011, ‘Bulldozer’; 3-operand, 256-bit regs, int. ops
- FMA4 – 2011, ‘Bulldozer’; 4-operand FMA

Agner Fog’s thoughts on the subject

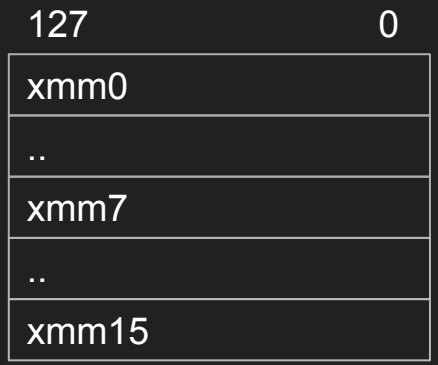
Status quo of SIMD across AMD64/x86-64

SSE2 is the baseline, but..

- Older entry-level AMD parts do SSSE3 + SSE4a (Bobcat)
- Entry-level Intel parts do SSE4.2 (Silvermont)
- Older high-end Intel parts do AVX (Sandy Bridge)
- New entry-level AMDs also do AVX (Jaguar)
- New mid-level Intels do AVX2 (Haswell)
- Only high-end (Xeon) Intels will do AVX-512

SSE2-SSE4.2 programming model

register file – 16 128-bit registers
(8 registers in 32-bit mode)



register layout, determined by the op:

- 4x fp32, 2x fp64
- 16x int8, 8x int16, 4x int32, 2x int64

scalar form of FP ops, where only $lane_0$ gets processed
(goodbye, venerable x87)

AVX-AVX2 programming model

register file – 16 256-bit regs
(8 registers in 32-bit mode)

AVX128 uses xmm, resets
bits 255..128 of corresp. ymm

AVX256 uses full-width ymm;

register layout same as in AVX128/SSE2, but **x2** as wide

255	127 0
ymm0	xmm0
..	..
ymm7	xmm7
..	..
ymm15	xmm15

SIMD register width vs SIMD throughput

Caveat emptor: AVX256 is just a programming model – not all ALU ops currently have 256-bit performance. It's the same as with SSEx before Merom, when select '128-bit' SSE ops had increased (as much as doubled) latency compared to their scalar counterparts.

Protip: read [5] and Agner Fog's indispensible http://www.agner.org/optimize/microarchitecture.pdf

SIMD intrinsics

Native C/C++ SIMD data types:

- `__m128`, `__m256` – 128- or 256-bit vector of fp32
- `__m128d`, `__m256d` – 128- or 256-bit vector of fp64
- `__m128i`, `__m256i` – 128- or 256-bit vector of an int type

Ops: all SIMD ops have intrinsics (see the ISA manual), but some intrinsics are nops (e.g. type casts)

https://software.intel.com/sites/landingpage/IntrinsicsGuide/

SIMD intrinsics, cont'd

Enablement: compiler targets and various headers:

- SSE2 – emmintrin.h
- SSE3 – pmmintrin.h
- SSE4.1 – smmintrin.h
- SSE4.2 – nmmintrin.h / smmintrin.h
- AVX – avxintrin.h
- AVX2 – avx2intrin.h

Or one target-aware common header: immintrin.h / x86intrin.h (unavailable with old compilers)

128-bit SIMD intrinsics

Same-functionality SSE and AVX128 intrinsics are identical and have 3-arg semantics, even though they represent 2-arg ops in SSE. This allows the compiler to generate AVX128 or SSEx code from the same intrinsics, depending on target.
/clever API engineering

Example:

```
__m128 c = _mm_add_ps(a, b); // same for SSE and AVX128
```

256-bit SIMD intrinsics

Example:

```
__m256 c = _mm256_add_ps(a, b); // AVX256 (AVX, AVX2)
```

There’s some good news and some bad news..

- Bad news: mixing SSE and AVX code is bad for performance.
- Good news: the compiler never does that - it generates only AVX128 and AVX256 from _mm_* and _mm256_* intrinsics for AVX-enabled targets; cross-library calls get special “guards” (vzeroupper AVX op).

The perils of mixing SSE and AVX

Going AVX→SSE incurs a (major) transition penalty, and here's what Intel has to say abou it:

AVX instructions always modify the upper bits of YMM registers and SSE instructions do not modify the upper bits. From a hardware perspective, the upper bits of the YMM register collection can be considered to be in one of three states:

• Clean: All upper bits of YMM are zero. This is the state when processor starts from RESET.
• Modified and saved to XSAVE region The content of the upper bits of YMM registers matches saved data in XSAVE region. This happens when after XSAVE/XRSTOR executes.
• Modified and Unsaved: The execution of one AVX instruction (either 256-bit or 128-bit) modifies the upper bits of the destination YMM.

The AVX/SSE transition penalty applies whenever the processor states is "Modified and Unsaved". Using VZEROUPPER move the processor states to "Clean" and avoid the transition penalty.

https://software.intel.com/en-us/forums/intel-performance-bottleneck-analyzer/topic/328259?language=it

SIMD operations overview

Op categories

- Loads/stores
- Arithmetic
- Relational
- Bitwise Logical
- Type Conversions
- Permutes, shuffles et al.

SSE/AVX128 arithmetics (excerpt)

```
__m128i _mm_add_epi8 (__m128i a, __m128i b);
__m128i _mm_add_epi16 (__m128i a, __m128i b);
__m128i _mm_add_epi32 (__m128i a, __m128i b);
__m128i _mm_add_epi64 (__m128i a, __m128i b);
__m128 _mm_add_ps (__m128 a, __m128 b);
__m128d _mm_add_pd (__m128d a, __m128d b);
__m128 _mm_add_ss (__m128 a, __m128 b);
__m128d _mm_add_sd (__m128d a, __m128d b);
```

SSE/AVX128 arithmetics (excerpt), cont'd

Add-with-saturate of small integer types (signed and unsigned):

```
__m128i _mm_adds_epi8 (__m128i a, __m128i b);
__m128i _mm_adds_epi16 (__m128i a, __m128i b);
__m128i _mm_adds_epu8 (__m128i a, __m128i b);
__m128i _mm_adds_epu16 (__m128i a, __m128i b);
...
```

https://software.intel.com/sites/landingpage/IntrinsicsGuide/

SSE/AVX128 relational

Vector and scalar ops:

- Vector relational ops return a vector mask, where each lane is either LaneType(-1) - true, or LaneType(0) - false. Single-lane versions copy over the upper lanes from the first argument.
- Scalar relational ops work with $lane_0$ alone and return a condition code of either 1 - true, or 0 - false.

SSE/AVX128 relational vector (excerpt)

```
__m128i _mm_cmpeq_epi8 (__m128i a, __m128i b);
__m128i _mm_cmpeq_epi16 (__m128i a, __m128i b);
__m128i _mm_cmpeq_epi32 (__m128i a, __m128i b);
__m128 _mm_cmpeq_ps (__m128 a, __m128 b);
__m128d _mm_cmpeq_pd (__m128d a, __m128d b);
__m128 _mm_cmpeq_ss (__m128 a, __m128 b);
__m128d _mm_cmpeq_sd (__m128d a, __m128d b);
...
```

SSE/AVX128 relational vector (excerpt), cont'd

Vector relational FP ops can check for the following conditions:

EQ / NEQ (equal / non-equal)

LT / NLT (less-than / non-less-than)

LE / NLE (less-or-equal / non-less-or-equal)

ORD / UNORD (ordered / unordered)

SSE/AVX128 relational scalar (excerpt)

```
int _mm_comieq_ss (__m128 a, __m128 b);
int _mm_comige_ss (__m128 a, __m128 b);
int _mm_comigt_ss (__m128 a, __m128 b);
int _mm_comile_ss (__m128 a, __m128 b);
int _mm_comilt_ss (__m128 a, __m128 b);
int _mm_comineq_ss (__m128 a, __m128 b);
...
```

SSE/AVX128 relational - please, note

Relational ops are numerous but, please, note:

- Every vector relational FP op has its inverse op (why?)
- Special attention to NaNs:
  - _mm_cmpord_* (vector), _mm_comi_* (scalar)
  - _mm_cmpunord_* (vector), _mm_ucomi_* (scalar)

SSE/AVX128 relational - NaNs

- Vector: ‘The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.’
- Scalar: ‘The UCOMISS instruction differs from the COMISS instruction in that it signals a SIMD floating-point invalid operation exception [...] only when a source operand is an SNaN. The COMISS instruction signals an invalid operation exception if a source operand is either a QNaN or an SNaN.’

SSE/AVX128 bitwise logical

AND, AND-NOT, OR, XOR

- Bitwise ops don’t care about NaNs - those are just bit patterns.
- No scalar versions - no reason for those.
- Bitwise ops exist in fp32, fp64 and int flavors; while fp64 is an artifact of the fp32 version, the int version is a different-pipeline instruction - beware!

SSE/AVX128 bitwise logical, cont'd

Lane 'true/false' extraction:

```
int _mm_movemask_epi8 (__m128i a);
int _mm_movemask_pd (__m128d a);
int _mm_movemask_ps (__m128 a);
```

Extract the MSB of each lane to an integer bitmask:

{ -1, 0, -1, 0 } → 1010_{bin} == 0xa == 10

How to use SIMD (from CG2 2015 Code-for-Art panel)

So you have your caches warm*, and your task has inherent parallelism - it’s time to use SIMD

- Autovectorization
- Intrinsics
- Generic vectors
- Libraries
  - Agner Fog's vector classes
  - embree simd classes

* Or the prefetchers are friendly to your access patterns.

Autovectorization

For instance, from..

```
void add(
    const float (& a)[16],
    const float (& b)[16],
    float (& res)[16]) {

    for (size_t i = 0; i < 16; ++i)
        res[i] = a[i] * b[i];
}
```

..the compiler could produce..

Autovectorization, cont'd

g++-5.1, AVX-enabled target

```
...
  4005cf:    vmovaps 0x2ae9(%rip),%ymm0            # 4030c0 <inputA>
  4005d7:    vmulps 0x2aa1(%rip),%ymm0,%ymm0       # 403080 <inputB>
  4005df:    vmovaps %ymm0,0x2b59(%rip)            # 403140 <res>
  4005e7:    vmovaps 0x2af1(%rip),%ymm0            # 4030e0 <inputA+0x20>
  4005ef:    vmulps 0x2aa9(%rip),%ymm0,%ymm0       # 4030a0 <inputB+0x20>
  4005f7:    vmovaps %ymm0,0x2b61(%rip)            # 403160 <res+0x20>
...
```

8-way SIMD load, multiplication and store (sizeof(8x_fp32) == 32 bytes)

Autovectorization, cont'd

clang++-3.6, AVX-enabled target

```
...
  400794:    vmovaps 0x1924(%rip),%xmm0          # 4020c0 <inputB>
  40079c:    vmulps 0x18dc(%rip),%xmm0,%xmm0     # 402080 <inputA>
  4007a4:    vmovaps %xmm0,0x1994(%rip)          # 402140 <res>
  4007ac:    vmovaps 0x191c(%rip),%xmm0          # 4020d0 <inputB+0x10>
  4007b4:    vmulps 0x18d4(%rip),%xmm0,%xmm0     # 402090 <inputA+0x10>
  4007bc:    vmovaps %xmm0,0x198c(%rip)          # 402150 <res+0x10>
  4007c4:    vmovaps 0x1914(%rip),%xmm0          # 4020e0 <inputB+0x20>
  4007cc:    vmulps 0x18cc(%rip),%xmm0,%xmm0     # 4020a0 <inputA+0x20>
  4007d4:    vmovaps %xmm0,0x1984(%rip)          # 402160 <res+0x20>
  4007dc:    vmovaps 0x190c(%rip),%xmm0          # 4020f0 <inputB+0x30>
  4007e4:    vmulps 0x18c4(%rip),%xmm0,%xmm0     # 4020b0 <inputA+0x30>
  4007ec:    vmovaps %xmm0,0x197c(%rip)          # 402170 <res+0x30>
...
```

4-way SIMD load, multiplication and store (sizeof(4x_fp32) == 16 bytes)

Intrinsics

From the same function written using intrinsics..

```
void add(
     const float (& a)[16],
     const float (& b)[16],
     float (& res)[16]) {

     _mm256_store_ps(res + 0, _mm256_mul_ps(_mm256_load_ps(a + 0), _mm256_load_ps(b + 0)));
     _mm256_store_ps(res + 8, _mm256_mul_ps(_mm256_load_ps(a + 8), _mm256_load_ps(b + 8)));
}
```

..the compiler must produce (given an AVX target)..

Intrinsics, cont’d

clang++-3.6, AVX-enabled target

```
...
  4006c4:    vmovaps 0x19b4(%rip),%ymm0          # 402080 <inputA>
  4006cc:    vmulps 0x19ec(%rip),%ymm0,%ymm0     # 4020c0 <inputB>
  4006d4:    vmovaps %ymm0,0x1a64(%rip)          # 402140 <res>
  4006dc:    vmovaps 0x19bc(%rip),%ymm0          # 4020a0 <inputA+0x20>
  4006e4:    vmulps 0x19f4(%rip),%ymm0,%ymm0     # 4020e0 <inputB+0x20>
  4006ec:    vmovaps %ymm0,0x1a6c(%rip)          # 402160 <res+0x20>
...
```

8-way SIMD load, multiplication and store

Generic vectors

Plain-old-data (POD) types with some special vector sauce..

```
typedef __attribute__ ((vector_size( 4 * sizeof(float)))) float vect4_float;

vect4_float a = (vect4_float) { 1.f, 2.f, 3.f, 4.f }; // literal
vect4_float b = (vect4_float) { 2.f, 1.f, 2.f, 1.f }; // literal

vect4_float res = a * b;
```

We do the vectorisation for the compiler, so it is free to produce SIMD code for an arbitrary vector architecture, or even for a scalar one – de-vectorisation is easy!

Generic vectors – some history

- Introduced by gcc in the early ‘00
- A generalization based on the popular SIMD architectures of the day (PowerPC AltiVec, x86 SSE), all the way to the ABI level
- Only partially supported by g++ until recently (full support in gcc) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=51033
- Fully supported by llvm/clang IR vectors - for llvm generic vectors are just another vector generalization at the frontent http://clang.llvm.org/docs/LanguageExtensions.html#vectors-and-extended-vectors
- Microsoft.. ‘will implement them when they become standard’ https://connect.microsoft.com/VisualStudio/feedback/details/804680/msvc-builtin-native-generic-cpu-agnostic-vector-support
  [update: above feedback ticket has since been purged from the msvc feedback db, but some traces still remain]

Generic vectors – matmul4x4 ‘hello-world’ example

Average performance of matrix 4x4 multiplicaions, expressed in FLOPs/clock, measured on an Intel **Ivy Bridge** under Ubuntu 14.04:

“Standard” method, scalar version:		“Standard” method, AVX 4-way intrinsics:		“Wide” method, 16-way generic vectors:		Via dot-products, AVX 4-way intrinsics	
g++-5.1.0:	1.566	g++-5.1.0:	6.858	g++-5.1.0:	0.164	g++:	3.436
clang++-3.6.2:	6.734	clang++-3.6.2:	6.874	clang++-3.6.2:	8.638	clang++:	3.398

- clang successfully autovectorises the scalar version (4-way SIMD + mul/add co-issue) - good!
- Intrinsics produce equal results across the compilers (4-way SIMD + mul/add co-issue).
- clang successfully uses AVX256 for the “wide” version (8-way SIMD + minor mul/add co-issue). Technically, so does g++, but at the cost of pathological permutations and spills of the arguments - the final performance is 1/10 of the scalar version!
- The dot-product version (with pre-transposed second argument) yields compact code, but just ½ of the performance of the reduction-free code.

https://github.com/ChaosGroup/cg2_2014_demo/blob/master/common/testvect_simd.cpp

https://github.com/ChaosGroup/cg2_2014_demo/blob/master/common/build_testvect_simd.sh

matmul4x4 example, cont’d

Average performance of matrix 4x4 multiplicaions, expressed in FLOPs/clock, measured on an Intel **Haswell** (successor to Intel Ivy Bridge):

“Wide” method,
16-way generic vectors:

clang++-3.6.2: 9.161

- clang achieves >9 FLOPs/clock vs 8.638 on Ivy Bridge - interesting how. The newer CPU must have a notably better IPC, or perhaps..

matmul4x4 example, cont'd

Ivy Bridge – AVX, no FMA3 ISA extension:

```
...
 4026c8:      vmulps %ymm4,%ymm3,%ymm3
 4026cc:      vmulps %ymm4,%ymm2,%ymm2
 4026d0:      vmulps %ymm7,%ymm6,%ymm4
 4026d4:      vmulps %ymm7,%ymm5,%ymm5
 4026d8:      vaddps %ymm5,%ymm2,%ymm2
 4026dc:      vaddps %ymm4,%ymm3,%ymm3
 4026e0:      vmulps %ymm10,%ymm9,%ymm4
 4026e5:      vmulps %ymm10,%ymm8,%ymm5
 4026ea:      vaddps %ymm5,%ymm3,%ymm3
 4026ee:      vaddps %ymm4,%ymm2,%ymm2
 4026f2:      vmulps %ymm11,%ymm1,%ymm1
 4026f7:      vmulps %ymm11,%ymm0,%ymm0
 4026fc:      vaddps %ymm0,%ymm2,%ymm0
 402700:      vaddps %ymm1,%ymm3,%ymm1
 402704:      vmovaps %ymm1,0x20(%rbx,%rcx,1)
 40270a:      vmovaps %ymm0,(%rbx,%rcx,1)
...
```

Haswell – AVX, FMA3 ISA extension:

```
...
 4024b8:      vmulps %ymm7,%ymm6,%ymm6
 4024bc:      vmulps %ymm7,%ymm5,%ymm5
 4024c0:      vfmadd213ps %ymm5,%ymm4,%ymm3
 4024c5:      vfmadd213ps %ymm6,%ymm4,%ymm2
 4024ca:      vfmadd213ps %ymm2,%ymm10,%ymm9
 4024cf:      vfmadd213ps %ymm3,%ymm10,%ymm8
 4024d4:      vfmadd213ps %ymm8,%ymm11,%ymm0
 4024d9:      vfmadd213ps %ymm9,%ymm11,%ymm1
 4024de:      vmovaps %ymm1,0x603760(%rcx)
 4024e6:      vmovaps %ymm0,0x603740(%rcx)
...
```

matmul4x4 example, cont'd

matmul loop iteration,
clang++-3.6, AVX + FMA3 target:

```
402440:   vmovaps 0x6036c0(%rcx),%ymm0
402448:   vmovaps 0x6036e0(%rcx),%ymm1
402450:   vpermilps $0x0,%ymm1,%ymm2
402456:   vpermilps $0x0,%ymm0,%ymm3
40245c:   vmovaps 0x603700(%rcx),%xmm4
402464:   vinsertf128 $0x1,%xmm4,%ymm4,%ymm4
40246a:   vpermilps $0x55,%ymm0,%ymm5
402470:   vpermilps $0x55,%ymm1,%ymm6
402476:   vmovaps 0x603710(%rcx),%xmm7
40247e:   vinsertf128 $0x1,%xmm7,%ymm7,%ymm7
402484:   vpermilps $0xaa,%ymm0,%ymm8
40248a:   vpermilps $0xaa,%ymm1,%ymm9
402490:   vmovaps 0x603720(%rcx),%xmm10
402498:   vinsertf128 $0x1,%xmm10,%ymm10,%ymm10
40249e:   vpermilps $0xff,%ymm1,%ymm1
4024a4:   vpermilps $0xff,%ymm0,%ymm0
```

computation (8-way SIMD)

```
4024aa:   vmovaps 0x603730(%rcx),%xmm11
4024b2:   vinsertf128 $0x1,%xmm11,%ymm11,%ymm11
4024b8:   vmulps %ymm7,%ymm6,%ymm6
4024bc:   vmulps %ymm7,%ymm5,%ymm5
4024c0:   vfmadd213ps %ymm5,%ymm4,%ymm3
4024c5:   vfmadd213ps %ymm6,%ymm4,%ymm2
4024ca:   vfmadd213ps %ymm2,%ymm10,%ymm9
4024cf:   vfmadd213ps %ymm3,%ymm10,%ymm8
4024d4:   vfmadd213ps %ymm8,%ymm11,%ymm0
4024d9:   vfmadd213ps %ymm9,%ymm11,%ymm1
4024de:   vmovaps %ymm1,0x603760(%rcx)
4024e6:   vmovaps %ymm0,0x603740(%rcx)
4024ee:   add    %rax,%rcx
4024f1:   add    $0xffffffffffffffff,%rdx
4024f5:   jne    402440 <main+0x220>
```

matmul4x4 example, cont'd

matmul loop iteration,
clang++-3.5, AArch64 target:

```
402418:    add    x11, x22, x8
40241c:    ldp    q0, q1, [x11]
402420:    ldp    q2, q3, [x11,#32]
402424:    ldp    q4, q5, [x11,#64]
402428:    ldp    q6, q7, [x11,#96]
40242c:    add    x11, x23, x8
402430:    sub    x10, x10, #0x1
402434:    fmul   v16.4s, v5.4s, v3.s[1]
402438:    fmul   v17.4s, v5.4s, v2.s[1]
40243c:    fmul   v18.4s, v5.4s, v1.s[1]
402440:    fmul   v5.4s, v5.4s, v0.s[1]
402444:    fmla   v5.4s, v4.4s, v0.s[0]
402448:    fmla   v18.4s, v4.4s, v1.s[0]
40244c:    fmla   v17.4s, v4.4s, v2.s[0]
402450:    fmla   v16.4s, v4.4s, v3.s[0]
```

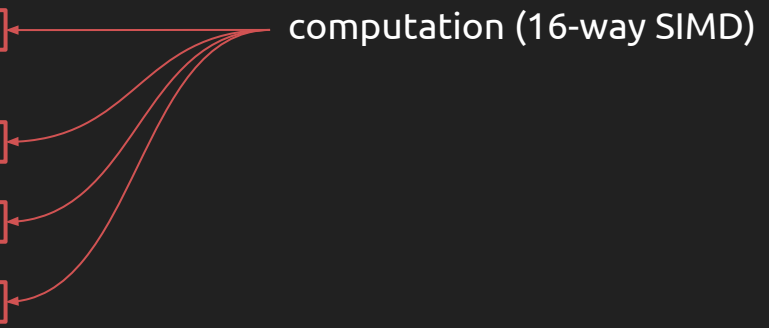
```
402454:    fmla   v16.4s, v6.4s, v3.s[2]
402458:    fmla   v17.4s, v6.4s, v2.s[2]
40245c:    fmla   v18.4s, v6.4s, v1.s[2]
402460:    fmla   v5.4s, v6.4s, v0.s[2]
402464:    fmla   v5.4s, v7.4s, v0.s[3]
402468:    fmla   v18.4s, v7.4s, v1.s[3]
40246c:    fmla   v17.4s, v7.4s, v2.s[3]
402470:    fmla   v16.4s, v7.4s, v3.s[3]
402474:    stp    q17, q16, [x11,#32]
402478:    stp    q5, q18, [x11]
40247c:    add    x8, x8, x9
402480:    cbnz   x10, 402418 <main+0x2b4>
```

computation (4-way SIMD)

matmul4x4 example, cont'd

matmul loop iteration, Knights Corner (KNC) intrinsics
icpc 14.0.4, MIC target:

```
4030d0:     vmovaps 0x604cc0(%rax),%zmm4
4030da:     vpermf32x4 $0x0,0x604d00(%rdx),%zmm0
4030e5:     vpermf32x4 $0x55,0x604d00(%rdx),%zmm1
4030f0:     vmulps %zmm4{aaaa},%zmm0,%zmm5
4030f6:     inc    %ecx
4030f8:     vpermf32x4 $0xaa,0x604d00(%rdx),%zmm2
403103:     vfmadd231ps %zmm4{bbbb},%zmm1,%zmm5
403109:     vpermf32x4 $0xff,0x604d00(%rdx),%zmm3
403114:     vfmadd231ps %zmm4{cccc},%zmm2,%zmm5
40311a:     add    %rsi,%rdx
40311d:     vfmadd231ps %zmm4{dddd},%zmm3,%zmm5
403123:     vmovnrngoaps %zmm5,0x604d40(%rax,%r14,1)
40312e:     add    %rsi,%rax
403131:     cmp    $0x3938700,%ecx
403137:     jb     4030d0 <compute(void*)+0xe0>
```



matmul4x4 example, multi-arch

Average performance of matrix 4x4 multiplicaions, expressed in FLOPs/clock, measured across:

Ivy Bridge (AVX256), 16-way generic vectors:		Haswell (AVX256 + FMA3), 16-way generic vectors:		Knights Corner (KNC), 16-way intrinsics:		Apple A8 (NEON), 4-way intrinsics	
clang++-3.5.2:	8.638	clang++-3.6.2:	9.161	icpc 14.0.4:	6.616	clang++7:	12.190

- The efficiency of SIMD greatly depends on the volume of permutes we do per unit of ALU work.
- The wider the SIMD, the more often we need to permute, on the average.
- Wide SIMD architectures require intelligent design of the ops - OoO won't help when you get data dependencies.
- ALU ops with built-in permute capabilities can give better results than sequences of permutes and ALU. Why?
- The ability to exclude (mask off) lanes in wide vectors is paramount for efficiency. AVX-512 (tentative name) inherits that trait from KNC.
- Despite its smart SIMD ISA, KNC fails due its hard-to-work-with uarch. Loop unrolling might help.
- Apple A8 NEON: good uarch + intelligent SIMD ISA = good performance.

References

[1] https://software.intel.com/sites/landingpage/IntrinsicsGuide
[2] http://www.naic.edu/~phil/software/intel/319433-014.pdf
[3] http://infocenter.arm.com/help/topic/com.arm.doc.ihi0073a/IHI0073A_arm_neon_intrinsics_ref.pdf
[4] https://www.element14.com/community/servlet/JiveServlet/previewBody/41836-102-1-229511/ARM.Reference_Manual.pdf
[5] http://www.agner.org/optimize/instruction_tables.pdf
[6] https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf
[7] http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf