Advanced MIC Programming

J. Eitzinger

PRACE PATC, 28.6.2017
Performance Engineering Tasks: Software side

Optimizing software for a specific hardware requires to align several orthogonal targets.

**Software side:** Reduce algorithmic and processor work

1. Reduce algorithmic work
2. Minimize processor work

Processor work consists of:
- **Instruction execution**
- **Data transfers**
Performance Engineering Tasks: Hardware

3. Distribute work and data for optimal utilization of parallel resources

4. Avoid bottlenecks

5. Use most effective execution units on chip
# Technologies Driving Performance

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIMD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock</td>
<td>33</td>
<td>200</td>
<td>1.1</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multicore</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **ILP Obstacle**: Not more parallelism available
- **Clock Obstacle**: Heat dissipation
- **Multi- Manycore Obstacle**: Getting data to/from cores

## Flavors of improvements
- Pure speed increase: Clock
- **Transparent** solutions
- **Explicit** solutions

## Strategies
- **Parallelism**
- **Specialisation**
History of Intel hardware developments

- Trade cores for frequency
The real picture
Finding the right compromise

- Core complexity
- Intel Skylake-EP
- Intel KNL
- Nvidia GP100

# cores
SIMD
Frequency
Maximum DP floating point (FP) performance

\[ P_{\text{core}} = n_{\text{super}}^{FP} \cdot n_{\text{FMA}} \cdot n_{\text{SIMD}} \cdot f \]

<table>
<thead>
<tr>
<th>uArch</th>
<th>( n_{\text{super}}^{FP} )</th>
<th>( n_{\text{FMA}} )</th>
<th>( n_{\text{SIMD}} )</th>
<th>( n_{\text{cores}} )</th>
<th>Release</th>
<th>Model</th>
<th>( P_{\text{core}} ) [GF/s]</th>
<th>( P_{\text{chip}} ) [GF/s]</th>
<th>( P_{\text{serial}} ) [GF/s]</th>
<th>TDP</th>
<th>GF/Watt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sandy Bridge</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>Q1/2012</td>
<td>E5-2680</td>
<td>11.7</td>
<td>173</td>
<td>7</td>
<td>130</td>
<td>1,33</td>
</tr>
<tr>
<td>Ivy Bridge</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>10</td>
<td>Q3/2013</td>
<td>E5-2690-v2</td>
<td>24</td>
<td>240</td>
<td>7,2</td>
<td>130</td>
<td>1,85</td>
</tr>
<tr>
<td>KNC</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>61</td>
<td>Q2/2014</td>
<td>7120A</td>
<td>10.6</td>
<td>1210</td>
<td>1,33</td>
<td>300</td>
<td>4,03</td>
</tr>
<tr>
<td>Haswell</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>14</td>
<td>Q3/2014</td>
<td>E5-2695-v3</td>
<td>21.6</td>
<td>425</td>
<td>6,6</td>
<td>120</td>
<td>3,54</td>
</tr>
<tr>
<td>Broadwell</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>22</td>
<td>Q1/2016</td>
<td>E5-2699-v4</td>
<td>17.6</td>
<td>704</td>
<td>7,2</td>
<td>145</td>
<td>4,85</td>
</tr>
<tr>
<td>Pascal</td>
<td>1</td>
<td>2</td>
<td>32</td>
<td>56</td>
<td>Q2/2016</td>
<td>GP100</td>
<td>36.8</td>
<td>4700</td>
<td>1,5</td>
<td>300</td>
<td>15,67</td>
</tr>
<tr>
<td>KNL</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>72</td>
<td>Q4/2016</td>
<td>7290F</td>
<td>35.2</td>
<td>2995</td>
<td>3,4</td>
<td>260</td>
<td>11,52</td>
</tr>
<tr>
<td>Skylake</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>26</td>
<td>Q3/2017</td>
<td>8170</td>
<td>23.4</td>
<td>1581</td>
<td>7,6</td>
<td>165</td>
<td>9,58</td>
</tr>
</tbody>
</table>
HARDWARE OPTIMIZATIONS FOR SINGLE-CORE EXECUTION

- SIMD
- SMT
- Memory hierarchy
KNL architecture

Core can retire 2 instructions per cycle
Core details: Simultaneous multi-threading (SMT)
Recommendations for data structure layout

- Promote temporal and spatial locality
- Enable packed (block wise) load/store of data
- Memory locality (placement)
- Avoid false cache line sharing
- Access data in long streams to enable efficient latency hiding

Above requirements may collide with object oriented programming paradigm: **array of structures** vs **structure of arrays**
## Comparison memory hierarchies

<table>
<thead>
<tr>
<th></th>
<th>Intel Broadwell-EP</th>
<th>Intel Xeon Phi KNL</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 D-Cache</td>
<td>32 kB</td>
<td>32 kB</td>
</tr>
<tr>
<td>L2</td>
<td>256 kB</td>
<td>1 MB shared</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32 MB total</td>
</tr>
<tr>
<td>L3</td>
<td>18 x 2.5 MB</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>45 MB total (shared)</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>4 channels DDR4-2400</td>
<td>6 channels DDR4-2133</td>
</tr>
<tr>
<td>Secondary Memory</td>
<td>-</td>
<td>16 GB MCDRAM</td>
</tr>
<tr>
<td>Peak Bandwidth</td>
<td>76.8 GB/s</td>
<td>102 GB/s, 450 GB/s</td>
</tr>
<tr>
<td>Update Bandwidth</td>
<td>98 GB/s (81%)</td>
<td>168 GB/s (53%)</td>
</tr>
</tbody>
</table>

Further differences:
- LLC on Xeon Phi is not shared
- Different MCDRAM modes are available: cache, flat, hybrid
- Latency DDR ca. 125ns and MCDRAM ca. 150ns
PARALLEL RESOURCES
SIMD
SIMD processing – Basics

- **Single Instruction Multiple Data (SIMD)** operations allow the **concurrent execution** of the **same operation** on “wide” registers.

- x86 SIMD instruction sets:
  - AVX-512: register width = 512 Bit → 8 DP floating point operands
  - AVX: register width = 256 Bit → 4 DP floating point operands

- Adding two registers holding double precision floating point operands

![Diagram showing SIMD and scalar execution examples](image)
Data types in 32-byte SIMD registers

- Supported data types depend on actual SIMD instruction set
real, dimension(:) :: A,B,C,D,R
SIMD processing – Basics

- Steps (done by the compiler) for “SIMD processing”

```c
for(int i=0; i<n;i++)
    C[i]=A[i]+B[i];
```

“Loop unrolling”

```c
for(int i=0; i<n;i+=4){
    C[i] =A[i] +B[i];
    C[i+1]=A[i+1]+B[i+1];
}
```

//remainder loop handling

```
Label1:
    VLOAD R0 ← A[i]
    VLOAD R1 ← B[i]
    V64ADD[R0,R1] → R2
    VSTORE R2 → C[i]
    i ← i+4
    i<(n-4)? JMP Label1
```

Load 256 Bits starting from address of A[i] to register R0

Add the corresponding 64 Bit entries in R0 and R1 and store the 4 results to R2

Store R2 (256 Bit) to address starting at C[i]

//remainder loop handling
SIMD processing – Basics

- No SIMD vectorization for loops with data dependencies:

```
for(int i=0; i<n;i++)
    A[i]=A[i-1]*s;
```

- “Pointer aliasing” may prevent SIMDfication

```c
void scale_shift(double *A, double *B, double *C, int n) {
    for(int i=0; i<n; ++i)
        C[i] = A[i] + B[i];
}
```

- C/C++ allows that $A \rightarrow &C[-1]$ and $B \rightarrow &C[-2]
  \rightarrow C[i] = C[i-1] + C[i-2]$: dependency $\rightarrow$ No SIMD

- If “pointer aliasing” is not used, tell it to the compiler, e.g. use
  `-fno-alias` switch for Intel compiler $\rightarrow$ SIMD
Why and how?

Why check the assembly code?

- Sometimes the only way to make sure the compiler “did the right thing”
  - Example: “LOOP WAS VECTORIZED” message is printed, but Loads & Stores may still be scalar!
- Get the assembler code (Intel compiler):
  - `icc -S -O3 triad.c -o a.out`
- Disassemble Executable:
  - `objdump -d ./a.out | less`

The x86 ISA is documented in:

- Intel Software Development Manual SDM
- Intel Architecture Instruction Set Extensions Programming Reference
Basics of the x86-64 ISA

- Instructions have 0 to 3 operands (4 with AVX-512)
- Operands can be registers, memory references or immediates
- Opcodes (binary representation of instructions) vary from 1 to 17 bytes
- There are two assembler syntax forms: Intel (left) and AT&T (right)
- Addressing Mode: BASE + INDEX * SCALE + DISPLACEMENT
- C: $A[i]$ equivalent to $*(A+i)$ (a pointer has a type: $A+i*8$)

```
movaps [rdi + rax*8+48], xmm3
add rax, 8
js 1b

movaps %xmm4, 48(%rdi,%rax,8)
addq $8, %rax
js ..B1.4

401b9f: 0f 29 5c c7 30    movaps %xmm3, 0x30(%rdi,%rax,8)
401ba4: 48 83 c0 08       add $0x8,%rax
401ba8: 78 a6             js 401b50 <triad_asm+0x4b>
```
Basics of the x86-64 ISA with extensions

16 general Purpose Registers (64bit):
rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8-r15
alias with eight 32 bit register set:
eax, ebx, ecx, edx, esi, edi, esp, ebp
8 opmask registers (16bit or 64bit, AVX512 only):
k0–k7

Floating Point **SIMD** Registers:

- xmm0–xmm15 (xmm31)  **SSE** (128bit) alias with 256-bit and 512-bit registers
- ymm0–ymm15 (xmm31)  **AVX** (256bit) alias with 512-bit registers
- zmm0–zmm31          **AVX-512** (512bit)

SIMD instructions are distinguished by:

- **VEX/EVEX** prefix:  \( \mathbf{v} \)
- Operation:           \( \text{mul, add, mov} \)
- Modifier:            \( \text{nontemporal (nt), unaligned (u), aligned (a), high (h)} \)
- Width:               \( \text{scalar (s), packed (p)} \)
- Data type:           \( \text{single (s), double (d)} \)
ISA support on KNL

KNL supports all *legacy* ISA extensions:

**MMX, SSE, AVX, AVX2**

Furthermore **KNL** supports:

- AVX-512 Foundation (F), KNL and Skylake
- AVX-512 Conflict Detection Instructions (CD), KNL and Skylake
- AVX-512 Exponential and Reciprocal Instructions (ER), KNL
- AVX-512 Prefetch Instructions (PF), KNL

AVX-512 extensions only supported on **Skylake**:

- AVX-512 Byte and Word Instructions (BW)
- AVX-512 Doubleword and Quadword Instructions (DQ)
- AVX-512 Vector Length Extensions (VL)

**ISA Documentation:**

*Intel Architecture Instruction Set Extensions Programming Reference*
Architecture specific issues KNC vs. KNL

KNC architectural issues

- Fragile single core performance (in-order, pairing, SMT)
- No proper hardware prefetching
- Shared access on segmented LLC costly

KNL fixes most of these issues and is more accessible!

Advises for KNL

- 1 thread per core is usually best, sometime two threads per core
- Large pages can improve performance significantly (2M, 1G)
- Consider the `-no-prec-div` option to enable AVX-512 ER instructions
- Aggressive software prefetching is usually not necessary
- MCDRAM is the preferred target memory (try cache mode first)
- Alignment restrictions and penalties are similar to Xeon. We experienced a benefit from alignment to page size with the MCDRAM.
Example for masked execution

Masking for predication is very helpful in cases such as e.g. remainder loop handling or conditional handling.
Gather instruction interface on KNC and Haswell

KNC:

kxnor k2, k2

..L100:

vgatherdps zmm13{k2}, [rdi + zmm17 * 4]
jkzd k2, ..L101
vgatherdps zmm13{k2}, [rdi + zmm17 * 4]
jknzd k2, ..L100

..L101:

Haswell:

vpcmpeqw ymm7, ymm7, ymm7
vgatherdps ymm15, [rdi + ymm11 * 4], ymm7
### Gather microbenchmarking results

<table>
<thead>
<tr>
<th></th>
<th>Knight Corner</th>
<th>Haswell</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1 Cache</td>
<td>L2 Cache</td>
</tr>
<tr>
<td></td>
<td>Instruction</td>
<td>Loop</td>
</tr>
<tr>
<td>16 per CL</td>
<td>9.0</td>
<td>9.0</td>
</tr>
<tr>
<td>8 per CL</td>
<td>4.2</td>
<td>8.4</td>
</tr>
<tr>
<td>4 per CL</td>
<td>3.7</td>
<td>14.8</td>
</tr>
<tr>
<td>2 per CL</td>
<td>2.9</td>
<td>23.2</td>
</tr>
<tr>
<td>1 per CL</td>
<td>2.3</td>
<td>36.8</td>
</tr>
</tbody>
</table>

**Serialization for loading several items per CL**

**No working prefetching for gather on KNC**

**KNL:**
- Implementation similar to Haswell
- Dedicated prefetch instructions available on KNL
Case Study: Simplest code for the summation of the elements of a vector (single precision)

```c
float sum = 0.0;

for (int j=0; j<size; j++){
    sum += data[j];
}
```

Instruction code:

```
401d08:   f3 0f 58 04 82  // addss xmm0,[rdx + rax * 4]
401d0d:   48 83 c0 01    // add    rax,1
401d11:   39 c7          // cmp     edi,eax
401d13:   77 f3          // ja      401d08
```

To get object code use `objdump -d` on object file or executable or compile with `-S`
Summation code (single precision): Optimizations

1:
addss xmm0, [rsi + rax * 4]
add rax, 1
cmp eax,edi
js 1b

3 cycles add pipeline latency

Unrolling with sub-sums to break up register dependency

1:
addps xmm0, [rsi + rax * 4]
addps xmm1, [rsi + rax * 4 + 16]
addps xmm2, [rsi + rax * 4 + 32]
addps xmm3, [rsi + rax * 4 + 48]
add rax, 16
cmp eax,edi
js 1b

SSE SIMD vectorization
SIMD influences instruction execution in the core – other runtime contributions stay the same!

AVX example (IvyBridge):

<table>
<thead>
<tr>
<th>Execution Units</th>
<th>2 cy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caches</td>
<td>4 cy</td>
</tr>
<tr>
<td>Memory</td>
<td>4 cy</td>
</tr>
</tbody>
</table>

Comparing total execution time:

<table>
<thead>
<tr>
<th></th>
<th>Execution</th>
<th>Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSE</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>AVX</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total runtime with data loaded from memory:

<table>
<thead>
<tr>
<th></th>
<th>Execution</th>
<th>Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSE</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVX</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Per-cacheline (8 iterations) cycle counts

SIMD only effective if runtime is dominated by instructions execution!
Summation code with AVX-512 (single core)

1:

```
vaddps zmm0, zmm0, [rsi + rax * 4]
vaddps zmm1, zmm1, [rsi + rax * 4 + 64]
vaddps zmm2, zmm2, [rsi + rax * 4 + 128]
vaddps zmm3, zmm3, [rsi + rax * 4 + 192]
add rax, 64
```

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>L2</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVX-512 plain SMT1</td>
<td>12942 MFlops/s 1.60 cycles/CL</td>
<td>12977 MFlops/s 1.60 cycles/CL</td>
<td>2256 MFlops/s 9.21 cycles/CL</td>
</tr>
<tr>
<td>AVX-512 plain SMT2</td>
<td>18101 MFlops/s 1.14 cycles/CL</td>
<td>12894 MFlops/s 1.61 cycles/CL</td>
<td>2976 MFlops/d 6.98 cycles/CL</td>
</tr>
<tr>
<td>IMCI plain SMT2</td>
<td>11863 MFlops/s 1.41 cycles/CL</td>
<td>14111 MFlops/s 11.85 cycles/CL</td>
<td>740 MFlops/s 22.64 cycles/CL</td>
</tr>
<tr>
<td>IMCI plain SMT4</td>
<td>10052 MFlops/s 1.66 cycles/CL</td>
<td>2730 MFlops/s 6.14 cycles/CL</td>
<td>904 MFlops/d 18.52 cycles/CL</td>
</tr>
</tbody>
</table>
A common technique to hide instruction latencies and loop overhead is deeper unrolling.

<table>
<thead>
<tr>
<th></th>
<th>KNC SMT2</th>
<th>KNL SMT1</th>
<th>KNL SMT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-way unrolled</td>
<td>11863 MFlops/s</td>
<td>12942 MFlops/s</td>
<td>18101 MFlops/s</td>
</tr>
<tr>
<td></td>
<td>1.41 cycles/CL</td>
<td>1.60 cycles/CL</td>
<td>1.14 cycles/CL</td>
</tr>
<tr>
<td>8-way unrolled</td>
<td>1.28 cycles/CL</td>
<td>24188 MFlops/s</td>
<td>22981 MFlops/s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.86 cycles/CL</td>
<td>0.91 cycles/CL</td>
</tr>
<tr>
<td>16-way unrolled</td>
<td><strong>1.21 cycles/CL</strong></td>
<td>29076 MFlops/s</td>
<td>27609 MFlops/s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.71 cycles/CL</td>
<td>0.75 cycles/CL</td>
</tr>
</tbody>
</table>

Peak is 1.3 Ghz * 2 instr/cycle * 16 Flops/instr = 41.6 Gflops/s (70%)
Pushing the limits: L2 performance

1:

```
vprefetch0 [rsi + rax * 4 + 256]
vaddps zmm0, zmm0, [rsi + rax * 4]
add rax, 16
cmp rax, rdi
jl 1b
```

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>L2</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-way unrolled</td>
<td>1.49 cycles/CL</td>
<td>6.03 cycles/CL</td>
<td>18.56 cycles/CL</td>
</tr>
<tr>
<td>SMT4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 prefetching</td>
<td>3.20 cycles/CL</td>
<td><strong>3.13 cycles/CL</strong></td>
<td>38.82 cycles/CL</td>
</tr>
<tr>
<td>SMT2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 prefetching</td>
<td>3.37 cycles/CL</td>
<td>3.85 cycles/CL</td>
<td>38.93 cycles/CL</td>
</tr>
<tr>
<td>SMT4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KNL 16-way unrolled</td>
<td>0.71 cycles/CL</td>
<td>1.53 cycles/CL</td>
<td>10.29 cycles/CL</td>
</tr>
<tr>
<td>SMT1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The software prefetching interferes with the hardware prefetcher.
### Shared L2 cache scalability

The L2 cache is shared by two cores.

<table>
<thead>
<tr>
<th></th>
<th>1 core</th>
<th>2 cores Shared L2</th>
<th>2 cores Private L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>KNL 16-way unrolled SMT1</td>
<td>53870 MFlops/s</td>
<td><strong>77598</strong> MFlops/s</td>
<td>107644 MFlops/s</td>
</tr>
</tbody>
</table>
Pushing the limits: Memory performance

```c
float sum=0.;
int i;

#pragma vector aligned
for(i = 0; i < length; i++)
{
    sum += A[i];
}
return sum;
```

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>L2</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-way unrolled</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMT4</td>
<td>1.49</td>
<td>6.03</td>
<td>18.56</td>
</tr>
<tr>
<td></td>
<td>cy/CL</td>
<td>cy/CL</td>
<td>cy/CL</td>
</tr>
<tr>
<td>L2 prefetching</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMT2</td>
<td>3.20</td>
<td>3.13</td>
<td>38.82</td>
</tr>
<tr>
<td></td>
<td>cy/CL</td>
<td>cy/CL</td>
<td>cy/CL</td>
</tr>
<tr>
<td>Memory prefetching</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMT2</td>
<td>3.05</td>
<td>4.98</td>
<td>14.17</td>
</tr>
<tr>
<td></td>
<td>cy/CL</td>
<td>cy/CL</td>
<td>cy/CL</td>
</tr>
<tr>
<td>KNL 16-way unrolled</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMT1 (MCDRAM)</td>
<td>0.71</td>
<td>1.53</td>
<td>10.92</td>
</tr>
<tr>
<td></td>
<td>cy/CL</td>
<td>cy/CL</td>
<td>(11.76)</td>
</tr>
</tbody>
</table>
# Summation code (full device)

<table>
<thead>
<tr>
<th>KNC</th>
<th>Single core</th>
<th>Full device</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 prefetching SMT2</td>
<td>1727 MB/s</td>
<td>90219 MB/s</td>
</tr>
<tr>
<td>MEM prefetching SMT1</td>
<td>4687 MB/s</td>
<td>170754 MB/s</td>
</tr>
<tr>
<td>MEM prefetching SMT2</td>
<td>4731 MB/s</td>
<td>175158 MB/s</td>
</tr>
<tr>
<td>MEM prefetching SMT4</td>
<td>4740 MB/s</td>
<td>176347 MB/s (62%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KNL</th>
<th>Single core</th>
<th>Full device</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMT1 DDR</td>
<td>8078 MB/s</td>
<td>78413 MB/s (76% Peak)</td>
</tr>
<tr>
<td>SMT1 MCDRAM</td>
<td>7072 MB/s</td>
<td>345198 MB/s</td>
</tr>
<tr>
<td>SMT2 MCDRAM</td>
<td>9443 MB/s</td>
<td>339352 MB/s</td>
</tr>
<tr>
<td>SMT4 MCDRAM</td>
<td>12363 MB/s</td>
<td>334483 MB/s</td>
</tr>
</tbody>
</table>

**MCDRAM**
- LLC on Xeon Phi is not shared
- Different MCDRAM modes are available: cache, flat, hybrid
How to leverage SIMD

Alternatives:
- The **compiler** does it for you (but: aliasing, alignment, language)
- Compiler directives (**pragmas**)
- Alternative **programming models** for compute kernels (OpenCL, cilk+, OpenMP 4, Intel ispc)
- C++ Vector classes
- **Intrinsics** (restricted to C/C++)
- Implement directly in **assembler**

To use **intrinsics** the following headers are available:
- `xmmintrin.h` (SSE)
- `pmmintrin.h` (SSE2)
- `immintrin.h` (AVX, AVX-512)
- `x86intrin.h` (all instruction set extensions)
- See next slide for an example
Example: array summation using C intrinsics (SSE, single precision)

```c
__m128 sum0, sum1, sum2, sum3;
__m128 t0, t1, t2, t3;
float scalar_sum;
sum0 = _mm_setzero_ps();
sum1 = _mm_setzero_ps();
sum2 = _mm_setzero_ps();
sum3 = _mm_setzero_ps();

for (int j=0; j<size; j+=16){
    t0 = _mm_loadu_ps(data+j);
    t1 = _mm_loadu_ps(data+j+4);
    t2 = _mm_loadu_ps(data+j+8);
    t3 = _mm_loadu_ps(data+j+12);
    sum0 = _mm_add_ps(sum0, t0);
    sum1 = _mm_add_ps(sum1, t1);
    sum2 = _mm_add_ps(sum2, t2);
    sum3 = _mm_add_ps(sum3, t3);
}

sum0 = _mm_add_ps(sum0, sum1);
sum0 = _mm_add_ps(sum0, sum2);
sum0 = _mm_add_ps(sum0, sum3);
sum0 = _mm_hadd_ps(sum0, sum0);
sum0 = _mm_hadd_ps(sum0, sum0);
_mm_store_ss(&scalar_sum, sum0);
```

summation of partial results
core loop (bulk)
Example: array summation from intrinsics, instruction code

14: 0f 57 c9 xorps %xmm1,%xmm1
17: 31 c0 xor %eax,%eax
19: 0f 28 d1 movaps %xmm1,%xmm2
1c: 0f 28 c1 movaps %xmm1,%xmm0
1f: 0f 28 d9 movaps %xmm1,%xmm3
22: 66 0f 1f 44 00 00 nopw 0x0(%rax,%rax,1)
28: 0f 10 3e movups (%rsi),%xmm7
2b: 0f 10 76 10 movups 0x10(%rsi),%xmm6
2f: 0f 10 6e 20 movups 0x20(%rsi),%xmm5
33: 0f 10 66 30 movups 0x30(%rsi),%xmm4
37: 83 c0 10 add $0x10,%eax
3a: 48 83 c6 40 add $0x40,%rsi
3e: 0f 58 df addps %xmm7,%xmm3
41: 0f 58 c6 addps %xmm6,%xmm0
44: 0f 58 d5 addps %xmm5,%xmm2
47: 0f 58 cc addps %xmm4,%xmm1
4a: 39 c7 cmp %eax,%edi
4c: 77 da ja 28 <compute_sum_SSE+0x18>
4e: 0f 58 c3 addps %xmm3,%xmm0
51: 0f 58 c2 addps %xmm2,%xmm0
54: 0f 58 c1 addps %xmm1,%xmm0
57: f2 0f 7c c0 haddps %xmm0,%xmm0
5b: f2 0f 7c c0 haddps %xmm0,%xmm0
5f: c3 retq
Example: array summation using C intrinsics (IMCI, single precision)

```c
float scalar_sum;
__m512 t0, t1, t2, t3;
__m512 sum0 = _mm512_setzero_ps();
__m512 sum1 = _mm512_setzero_ps();
__m512 sum2 = _mm512_setzero_ps();
__m512 sum3 = _mm512_setzero_ps();

for(i = 0; i < length; i+=64)
{
    t0 = _mm512_load_ps(data+i);
    t1 = _mm512_load_ps(data+i+16);
    t2 = _mm512_load_ps(data+i+32);
    t3 = _mm512_load_ps(data+i+48);
    sum0 = _mm512_add_ps(sum0, t0);
    sum1 = _mm512_add_ps(sum1, t1);
    sum2 = _mm512_add_ps(sum2, t2);
    sum3 = _mm512_add_ps(sum3, t3);
    sum0 = _mm512_add_ps(sum0, sum1);
    sum0 = _mm512_add_ps(sum0, sum2);
    sum0 = _mm512_add_ps(sum0, sum3);
    t0 = (__m512) _mm512_permute4f128_epi32((__m512i)sum0, _MM_PERM_DCDC);
    sum0 = _mm512_add_ps(sum0, t0);
    t1 = (__m512) _mm512_permute4f128_epi32((__m512i)sum0, _MM_PERM_BBBB);
    sum1 = _mm512_add_ps(sum1, _mm512_swizzle_ps(sum0, _MM_SWIZ_REG_BADC));
    sum2 = _mm512_add_ps(sum2, _mm512_swizzle_ps(sum1, _MM_SWIZ_REG_CDAB));
    _mm512_extpackstorelo_ps(&scalar_sum, sum2, _MM_DOWNCONV_PS_NONE, _MM_HINT_NONE);
}
```
Example: array summation from IMCI intrinsics, instruction code

..B2.3:

    vaddps (%rdi,%rdx,4), %zmm3, %zmm3
    vprefetch1 1024(%rdi,%rdx,4)
    vaddps 64(%rdi,%rdx,4), %zmm2, %zmm2
    vprefetch0 512(%rdi,%rdx,4)
    vaddps 128(%rdi,%rdx,4), %zmm1, %zmm1
    incl %ecx
    vaddps 192(%rdi,%rdx,4), %zmm0, %zmm0
    addq $64, %rdx
    cmpl %eax, %ecx
    jb ..B2.3

..B2.5:

    vaddps %zmm2, %zmm3, %zmm2
    vaddps %zmm1, %zmm2, %zmm1
    vaddps %zmm0, %zmm1, %zmm3
    nop
    vpermf32x4 $238, %zmm3, %zmm4
    vaddps %zmm4, %zmm3, %zmm5
    nop
    vpermf32x4 $85, %zmm5, %zmm6
    vaddps %zmm6, %zmm5, %zmm7
    nop
    vaddps %zmm7{badc}, %zmm7, %zmm8
    nop
    vaddps %zmm8{cdab}, %zmm8, %zmm9
    nop
    vpackstorelps %zmm9, -8(%rsp)
Vectorization and the Intel compiler

- Intel compiler will try to use SIMD instructions when enabled to do so
  - “Poor man’s vector computing”
  - Compiler can emit messages about vectorized loops (not by default):

    plain.c(11): (col. 9) remark: LOOP WAS VECTORIZED.

- Use option `-vec_report3` to get full compiler output about which loops were vectorized and which were not and why (data dependencies!)
- Some obstructions will prevent the compiler from applying vectorization even if it is possible

- You can use **source code directives** to provide more information to the compiler
Rules for vectorizable loops

1. Countable
2. Single entry and single exit
3. Straight line code
4. No function calls (exception intrinsic math functions)

Better performance with:
1. Simple inner loops with unit stride
2. Minimize indirect addressing
3. Align data structures (SSE 16 bytes, AVX 32 bytes)
4. In C use the restrict keyword for pointers to rule out aliasing

Obstacles for vectorization:
- Non-contiguos memory access
- Data dependencies
x86 Architecture:
SIMD and Alignment

- Alignment issues
  - Alignment of arrays with AVX (IMCI) should be on 32-byte (64-byte) boundaries to allow packed aligned loads and NT stores (for Intel processors)
  - Modern x86 CPUs have less (not zero) impact for misaligned LD/ST, but Xeon Phi relies heavily on it!
- How is manual alignment accomplished?
- Dynamic allocation of aligned memory (align = alignment boundary):

```c
#define _XOPEN_SOURCE 600
#include <stdlib.h>

int posix_memalign(void **ptr,
                    size_t align,
                    size_t size);
```
Interlude: Software Prefetching on Xeon Phi

- Compiler will issue a massive amount of prefetch instructions starting with –O2
- This includes all intrinsic load and stores
- This is a reasonable compromise to deal with the shortcomings of the overall architecture

- To turn off software prefetching by the compiler:
  - Global option –no-opt-prefetch
  - Loop local pragma #pragma noprefetch

- To be sure always check the assembly code, especially with Intrinsic code.
Probing of the memory hierarchy
Saturation effects in cache and memory
Typical OpenMP overheads
LLC performance on Xeon Phi (1 core)

![Graph showing LLC performance on Xeon Phi (1 core)](image)
LLC performance on SandyBridge-EP (1 core)
LLC bandwidth scaling Xeon Phi
LLC bandwidth scaling SandyBridge-EP
Memory bandwidth saturation on Xeon Phi

![Graph showing memory bandwidth saturation on Xeon Phi](image-url)
Memory bandwidth saturation on SandyBridge-EP

![Graph showing memory bandwidth saturation on SandyBridge-EP with different core counts, comparing Update (no SMT), Update (2-SMT), Copy (no SMT), and Copy (2-SMT).]
### Thread synchronization overhead on IvyBridge-EP

**Barrier overhead in CPU cycles**

<table>
<thead>
<tr>
<th></th>
<th>Intel 16.0</th>
<th>GCC 5.3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2 Threads</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared L3</td>
<td>599</td>
<td>425</td>
</tr>
<tr>
<td>SMT threads</td>
<td>612</td>
<td>423</td>
</tr>
<tr>
<td>Other socket</td>
<td>1486</td>
<td>1067</td>
</tr>
</tbody>
</table>

\[
\text{Strong topology dependence!}
\]

<table>
<thead>
<tr>
<th></th>
<th>Intel 16.0</th>
<th>GCC 5.3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full domain</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Socket (10 cores)</td>
<td>1934</td>
<td>1301</td>
</tr>
<tr>
<td>Node (20 cores)</td>
<td>4999</td>
<td>7783</td>
</tr>
<tr>
<td>Node +SMT</td>
<td>5981</td>
<td>9897</td>
</tr>
</tbody>
</table>

- **Strong dependence on compiler, CPU and system environment!**
- **OMP_WAIT_POLICY=ACTIVE** can make a big difference

\[
\text{Overhead grows with thread count}
\]
Thread synchronization overhead on Intel Xeon Phi KNC (60-core)

Barrier overhead in CPU cycles

<table>
<thead>
<tr>
<th></th>
<th>SMT1</th>
<th>SMT2</th>
<th>SMT3</th>
<th>SMT4</th>
</tr>
</thead>
<tbody>
<tr>
<td>One core</td>
<td>n/a</td>
<td>1597</td>
<td>2825</td>
<td>3557</td>
</tr>
<tr>
<td>Full chip</td>
<td>10604</td>
<td>12800</td>
<td>15573</td>
<td>18490</td>
</tr>
</tbody>
</table>

That does not look bad for 240 threads!

Still the pain may be much larger, as more work can be done in one cycle on Phi compared to a full Sandy Bridge node.

3.75 x cores (16 vs 60) on Phi
2 x more operations per cycle on Phi
2.7 x more barrier penalty (cycles) on Phi

7.5 x more work done on Xeon Phi per cycle

One barrier causes 2.7 x 7.5 = 20x more pain 😊.
Thread synchronization overhead on Xeon Phi KNL 7210 (64-core)

Barrier overhead in CPU cycles (Intel C compiler 16.03)

<table>
<thead>
<tr>
<th></th>
<th>SMT1</th>
<th>SMT2</th>
<th>SMT3</th>
<th>SMT4</th>
</tr>
</thead>
<tbody>
<tr>
<td>One core</td>
<td>n/a</td>
<td>963</td>
<td>1580</td>
<td>2240</td>
</tr>
<tr>
<td>Full chip</td>
<td>5720</td>
<td>8100</td>
<td>9900</td>
<td>11400</td>
</tr>
</tbody>
</table>

Still the pain may be much larger, as more work can be done in one cycle on Phi compared to a full Ivy Bridge node

3.2x cores (20 vs 64) on Phi
4x more operations per cycle per core on Phi

→ 4 · 3.2 = 12.8x more work done on Xeon Phi per cycle

1.9x more barrier penalty (cycles) on Phi (11400 vs. 6000)

→ One barrier causes 1.9 · 12.8 ≈ 24x more pain 😞.
Configuration complexity

- **Cluster modes**: lower the latency and increase the bandwidth
  - All-to-all
  - Quadrant mode (default)
  - Sub-numa-clustering (SNC), best performance but explicit

- **Memory modes**:
  - Cache mode (default)
  - Flat mode (explicit)
  - Hybrid

- **Mapping** of application on hardware:
  - Use SMT or not. How many SMT threads?
  - Use all cores?
  - MPI+X. How exactly?

- **Memory configuration**: Alignment and page size choices
Specific issues with Xeon Phi

- **MCDRAM** adds additional complexity

- Configuration of system and mapping of application on hardware gets more critical

- The compromise and strategy made with KNL will soon be outdated

- KNL as a hosted cluster system is probably too specialized for a general purpose academic cluster
But

- Xeon Phi implements features which are not available anywhere else:
  - High degree of parallelism
  - Multiple memory types and explicit memory control
  - Mesh type on-die topology

- It allowed a glimpse in the future on real hardware
Documentation

- Intel® Xeon Phi Processor High Performance Programming Knights Landing Edition By Jim Jeffers, James Reinders, Avinash Sodani

- Intrinsics guide as interactive webpage

- Intel® Architecture Instruction Set Extensions Programming Reference Document ID 319433-029 APRIL 2017