Advanced MIC Programming

J. Eitzinger

PRACE PATC, 29.4.2015
Common technologies

- Instruction Level Parallelism (ILP)
  - Instruction pipelining
  - Superscalar execution
  - Out-of-order execution

- Memory Hierarchy

- Branch Prediction Unit, Hardware Prefetching

- Single Instruction Multiple Data (SIMD)

- Simultaneous Multithreading (SMT)
Parallelism in a modern compute node

Parallel and shared resources within a shared-memory node

Parallel resources:
- Execution/SIMD units
- Cores
- Inner cache levels
- Sockets / ccNUMA domains
- Multiple accelerators

Shared resources:
- Outer cache level per socket
- Memory bus per socket
- Intersocket link
- PCIe bus(es)
- Other I/O resources

Exploiting performance: parallelism + bottleneck awareness
HARDWARE OPTIMIZATIONS FOR SINGLE-CORE EXECUTION

- SIMD
- SMT
- Memory hierarchy
Core details: Simultaneous multi-threading (SMT)

Standard core

Memory

L2 cache

L1D cache

L1I cache

Registers

Control

Execution units

2-way SMT

Memory

L2 cache

L1D cache

L1I cache

Registers

Control

Execution units
Single Instruction Multiple Data (SIMD) allows the concurrent execution of the same operation on “wide” registers.

- IMCI: 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

**Scalar execution:**
R2 ← ADD [R0,R1]

**SIMD execution:**
V64ADD [R0,R1] → R2
Latency and bandwidth in modern computer environments

Avoiding slow data paths is the key to most performance optimizations!
How does data travel from memory to the CPU and back?

Remember: Caches are organized in cache lines (e.g., 64 bytes) Only complete cache lines are transferred between memory hierarchy levels (except registers)

**MISS**: Load or store instruction does not find data in a cache level → CL transfer required

Example: Array copy \( A( : ) = C( : ) \)
Recap: Data transfers in a memory hierarchy

- How does data travel from memory to the CPU and back?
- Example: Array copy \( A(\:) = C(\:) \)

![Diagram showing data transfers and cache hits and misses.](image)
Consequences 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
The driving forces behind performance

\[ P = n_{\text{core}} \times F \times S \times \nu \]

<table>
<thead>
<tr>
<th></th>
<th>Intel IvyBridge-EP</th>
<th>IBM Power7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores (n_{\text{core}})</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>FP instructions per cycle (F)</td>
<td>2</td>
<td>2 (DP) / 1 (SP)</td>
</tr>
<tr>
<td>FP ops per instructions (S)</td>
<td>4 (DP) / 8 (SP)</td>
<td>2 (DP) / 4 (SP) - FMA</td>
</tr>
<tr>
<td>Clock speed [GHz] (\nu)</td>
<td>2.7</td>
<td>3.7</td>
</tr>
<tr>
<td>Performance [GF/s] (P)</td>
<td>259 (DP) / 518 (SP)</td>
<td>236 (DP/SP)</td>
</tr>
</tbody>
</table>

TOP500 rank 1 (1996)

But: \(P=5.4\) GF/s or \(14.8\) GF/s(dp) for serial, non-SIMD code
What is different on MIC?

### Intel Xeon Phi

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores ( n_{\text{core}} )</td>
<td>60</td>
</tr>
<tr>
<td>FP instructions per cycle ( F )</td>
<td>1</td>
</tr>
<tr>
<td>FP ops per instructions ( S )</td>
<td>2 (FMA) x 8 (DP) / 16 (SP)</td>
</tr>
<tr>
<td>Clock speed [GHz] ( \nu )</td>
<td>1.05</td>
</tr>
<tr>
<td>Performance [GF/s] ( P )</td>
<td>1008 (DP) / 2016 (SP)</td>
</tr>
</tbody>
</table>

**But:** \( P=1.05 \) GF/s for serial, non-simd code

- 5x number of cores
- 2x SIMD lanes
- Slower clock
Xeon Phi Core architecture

- In-order execution
- 2-way superscalar
- 4-way SMT
- Scalar V-Pipe
- 512bit Vector Unit U-Pipe
- 32 64byte Vector registers
- 8 16bit Vector Mask registers
Comparison memory hierarchies

<table>
<thead>
<tr>
<th></th>
<th>Intel IvyBridge-EP</th>
<th>Intel Xeon Phi</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>512 kB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30 MB total</td>
</tr>
<tr>
<td>L3</td>
<td>12 x 2.5 MB</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>30 MB total (shared)</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>8 channels DDR3-1866(2S node)</td>
<td>6 channels GDDR5 5GHz</td>
</tr>
<tr>
<td>Peak Bandwidth</td>
<td>119.4 GB/s</td>
<td>320 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:
- On Xeon Phi no hardware prefetchers between L1 and L2
- More sophisticated hardware prefetchers on IvyBridge-EP
- LLC on Xeon Phi is not shared. Cores can steal data from remote cache segments but not place data in them.
Conclusions about core architectures

- All efforts are targeted on increasing **instruction throughput**
- Every hardware optimization puts an **assumption** against the executed software
- One can distinguish transparent and **explicit** solutions

**Common technologies:**
- Instruction level parallelism (**ILP**)
- Data parallel execution (**SIMD**), does not affect instruction throughput
- Exploit temporal data access locality (**Caches**)
- Hide data access latencies (**Prefetching**)
- Avoid hazards
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:
  - IMCI: register width = 512 Bit → 8 double precision floating point operands
  - AVX: register width = 256 Bit → 4 double precision floating point operands

- Adding two registers holding double precision floating point operands

```
A[0] + B[0] -> C[0]
```
**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

**Loop unrolling**

- 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]

LABEL1:

```c
VLOAD R0 ← A[i]
VLOAD R1 ← B[i]
V64ADD[R0,R1] → R2
VSTORE R2 → C[i]
i←i+4
i<(n-4)? JMP LABEL1
```

//remainder loop handling
SIMD processing – Basics

- No SIMD vectorization for loops with data dependencies:

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

- “Pointer aliasing” may prevent SIMD-fication

```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 \( B \rightarrow &C[-2] \)
  \( C[i] = C[i-1] + C[i-2] \) : 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 Xeon Phi Coprocessor Instruction Set Reference Manual
Basics of the x86-64 ISA

- Instructions have 0 to 2 operands (3 with AVX)
- Operands can be registers, memory references or immediates
- Opcodes (binary representation of instructions) vary from 1 to 17 bytes
- There are two 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)

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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>401b9f</td>
<td>0f 29 5c c7 30</td>
<td>movaps %xmm3,0x30(%rdi,%rax,8)</td>
</tr>
<tr>
<td>401ba4</td>
<td>48 83 c0 08</td>
<td>add $0x8,%rax</td>
</tr>
<tr>
<td>401ba8</td>
<td>78 a6</td>
<td>js 401b50 &lt;triad_asm+0x4b&gt;</td>
</tr>
</tbody>
</table>
Basics of the x86-64 ISA

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

Floating Point SIMD Registers:
 xmm0–xmm15    SSE (128bit) alias with 256-bit registers
 ymm0–ymm15    AVX (256bit)

SIMD instructions are distinguished by:
AVX (VEX) prefix:  v
Operation:        mul, add, mov
Modifier:         nontemporal (nt), unaligned (u), aligned (a), high (h)
Width:            scalar (s), packed (p)
Data type:        single (s), double (d)
IMCI Vector instruction set extension

- No support for SSE and AVX/AVX2!
- No dedicated scalar instruction. Scalar code possible using masking.
- Fused multiply add instructions
- Gather/Scatter instructions

Floating Point **SIMD** Registers:

\texttt{zmm0-zmm31} (512bit)

Vector mask registers:

\texttt{k0-k7} (16bit)

SIMD instructions are distinguished by:

AVX (VEX) prefix: \texttt{v}

Operation: \texttt{mul, add, mov}

Modifier: unaligned (\texttt{u}), aligned (\texttt{a}), high (\texttt{h})

Data type: single (\texttt{s}), double (\texttt{d})
Example for masked execution

- **k1**: 16 bits
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:

```
addss xmm0,[rdx + rax * 4]
add rax,1
cmp edi,eax
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

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

Unrolling with sub-sums to break up register dependency

3 cycles add pipeline latency

SSE SIMD vectorization
SIMD processing – single-threaded

SIMD influences instruction execution in the core – other bottlenecks stay the same!

- **Full benefit in L1 cache**
- **Data transfers are overlapped with execution**

### Per-cache-line cycle counts
- **Execution**: 48, 16, 4, 4
- **Cache**: L1D 16 cycles, L1D 4 cycles, L2 2 cycles, L3 2 cycles
- **Memory**: 4 cycles

### Peak Mflops/s
- **Scalar**: 12500
- **Plain**: 15000
- **SIMD**: 15000

- **L1**: 16 cycles
- **L3**: 8 cycles
- **MEM**: 24 cycles

### Data transfers
- **Are overlapped with execution**
- **Some penalty for SIMD (12 cy predicted)**
And with AVX?

With preloading:
AVX down to less than 7 cycles (8309 MFlops/s)
SIMD processing – Full chip (all cores)

Influence of SMT

Bandwidth saturation is the primary performance limitation on the chip level!

Full scaling using SMT due to bubbles in pipeline

Conclusion: If the code saturates the bottleneck, all variants are acceptable!

All variants saturate the memory bandwidth

8 threads on physical cores

16 threads using SMT
Summation code with IMCI (single core)

1:

\[
\begin{align*}
vaddps & \ zmm0, \ zmm0, \ [rsi + \ rax \times 4] \\
vaddps & \ zmm1, \ zmm1, \ [rsi + \ rax \times 4 + 64] \\
vaddps & \ zmm2, \ zmm2, \ [rsi + \ rax \times 4 + 128] \\
vaddps & \ zmm3, \ zmm3, \ [rsi + \ rax \times 4 + 192] \\
add & \ rax, \ 64 \\
cmp & \ rax, \ rdi \\
jl & \ 1b \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>L2</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMCI plain SMT2</td>
<td>11863 MFlops/s</td>
<td>1411 MFlops/s</td>
<td>740 MFlops/s</td>
</tr>
<tr>
<td></td>
<td>1.41 cycles/CL</td>
<td>11.85 cycles/CL</td>
<td>22.64 cycles/CL</td>
</tr>
<tr>
<td>IMCI plain SMT4</td>
<td>10052 MFlops/s</td>
<td>2730 MFlops/s</td>
<td>904 MFlops/d</td>
</tr>
<tr>
<td></td>
<td>1.66 cycles/CL</td>
<td>6.14 cycles/CL</td>
<td>18.52 cycles/CL</td>
</tr>
</tbody>
</table>
Pushing the limits: L1 performance

A common technique to hide loop overhead is deeper unrolling.

Unrolling reduces the loop overhead impact

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>L2</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-way unrolled</td>
<td>1.41 cycles/CL</td>
<td>11.85 cycles/CL</td>
<td>22.64 cycles/CL</td>
</tr>
<tr>
<td>SMT2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-way unrolled</td>
<td>1.28 cycles/CL</td>
<td>11.75 cycles/CL</td>
<td>22.64 cycles/CL</td>
</tr>
<tr>
<td>SMT2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-way unrolled</td>
<td>1.21 cycles/CL</td>
<td>11.68 cycles/CL</td>
<td>22.64 cycles/CL</td>
</tr>
<tr>
<td>SMT2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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>
</tbody>
</table>

The software prefetching interferes with the hardware prefetcher.
Pushing the limits: Memory performance

1:
  vaddps zmm0, zmm0, [rsi + rax * 4]
  vprefetch1 [rsi + rax * 4 + 4096]
  vaddps zmm1, zmm1, [rsi + rax * 4 + 64]
  vprefetch0 [rsi + rax * 4 + 1024]
  vaddps zmm2, zmm2, [rsi + rax * 4 + 128]
  vprefetch1 [rsi + rax * 4 + 4160]
  vaddps zmm3, zmm3, [rsi + rax * 4 + 192]
  vprefetch0 [rsi + rax * 4 + 1088]
  vaddps zmm4, zmm4, [rsi + rax * 4 + 256]
  vprefetch1 [rsi + rax * 4 + 4224]
  vaddps zmm5, zmm5, [rsi + rax * 4 + 320]
  vprefetch0 [rsi + rax * 4 + 1152]
  vaddps zmm6, zmm6, [rsi + rax * 4 + 384]
  vprefetch1 [rsi + rax * 4 + 4288]
  vaddps zmm7, zmm7, [rsi + rax * 4 + 448]
  vprefetch0 [rsi + rax * 4 + 1216]
  vprefetch1 [rsi + rax * 4 + 4352]
  vprefetch0 [rsi + rax * 4 + 1280]
  vprefetch1 [rsi + rax * 4 + 4416]
  vprefetch0 [rsi + rax * 4 + 1344]
  vprefetch1 [rsi + rax * 4 + 4480]
  vprefetch0 [rsi + rax * 4 + 1408]
  vprefetch1 [rsi + rax * 4 + 4544]
  vprefetch0 [rsi + rax * 4 + 1472]
  add rax, 128
  cmp rax, rdi
  jl 1b

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 cy/CL</td>
<td>6.03 cy/CL</td>
<td>18.56 cy/CL</td>
</tr>
<tr>
<td>L2 prefetching</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMT2</td>
<td>3.20 cy/CL</td>
<td><strong>3.13 cy/CL</strong></td>
<td>38.82 cy/CL</td>
</tr>
<tr>
<td>Memory prefetching</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMT2</td>
<td>3.05 cy/CL</td>
<td>4.98 cy/CL</td>
<td><strong>14.17 cy/CL</strong></td>
</tr>
</tbody>
</table>

This is the default code generated by the compiler.
## Summation code with IMCI (full device)

<table>
<thead>
<tr>
<th></th>
<th>Single core</th>
<th>Full device</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMT1</td>
<td>1927 MB/s</td>
<td>95986 MB/s (83%)</td>
</tr>
<tr>
<td>SMT2</td>
<td>2962 MB/s</td>
<td>131814 MB/s (74%)</td>
</tr>
<tr>
<td>SMT3</td>
<td>3390 MB/s</td>
<td>146846 MB/s (72%)</td>
</tr>
<tr>
<td>SMT4</td>
<td>3620 MB/s</td>
<td>147159 MB/s (68%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></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 (87%)</td>
</tr>
<tr>
<td>MEM prefetching SMT1</td>
<td>4687 MB/s</td>
<td>170754 MB/s (60%)</td>
</tr>
<tr>
<td>MEM prefetching SMT2</td>
<td>4731 MB/s</td>
<td>175158 MB/s (62%)</td>
</tr>
<tr>
<td>MEM prefetching SMT4</td>
<td>4740 MB/s</td>
<td><strong>176347 MB/s (62%)</strong></td>
</tr>
</tbody>
</table>
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, ispc)
- **Intrinsics** (restricted to C/C++)
- Implement directly in **assembler**

To use **intrinsics** the following headers are available:

- `xmmmintrin.h` (SSE)
- `pmmintrin.h` (SSE2)
- `immintrin.h` (AVX, IMCI)
- `x86intrin.h` (all instruction set extensions)
- See next slide for an example
Example: array summation using C intrinsics (SSE, single precision)

```
__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);
_mm_store_ss(&scalar_sum, sum0);
```

summation of partial results

core loop (bulk)
**Example: array summation from intrinsics, instruction code**

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

**Loop body**
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 = (_mm512) _mm512_permute4f128_epi32(_mm512i)sum0, _MM_PERM_DCDC);
    sum0 = _mm512_add_ps(sum0, t0);
    t1 = (_mm512) _mm512_permute4f128_epi32(_mm512i)sum0, _MM_PERM_BBBB);
    sum0 = _mm512_add_ps(sum0, t1);
    sum1 = _mm512_add_ps(sum0, _mm512_swizzle_ps(sum0, _MM_SWIZ_REG_BADC));
    sum2 = _mm512_add_ps(sum1, _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)
```

Loop body
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-contiguous memory access
- Data dependencies
x86 Architecture: 
**SIMD and Alignment**

- **Alignment issues**
  - Alignment of arrays with SSE (AVX) should be on 16-byte (32-byte) boundaries to allow packed aligned loads and NT stores (**for Intel processors**)
    - AMD has a scalar non-temporal store instruction
  - Otherwise the compiler will revert to unaligned loads and not use NT stores – even if you say `vector nontemporal`
  - 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);
```

---

# FAU

---

42
How do we figure out the node topology?

**LIKWID** tool suite:

Like
I
Knew
What
I’m
Doing

Open source tool collection (developed at RRZE):
http://code.google.com/p/likwid

http://arxiv.org/abs/1004.4431
Likwid Tool Suite

- Command line tools for Linux:
  - easy to install
  - works with standard linux kernel
  - simple and clear to use
  - supports Intel and AMD CPUs

- Current tools:
  - **likwid-topology**: Print thread and cache topology
  - **likwid-pin**: Pin threaded application without touching code
  - **likwid-perfctr**: Measure performance counters
  - **likwid-powermeter**: Query turbo mode steps. Measure ETS.
  - **likwid-bench**: Low-level bandwidth benchmark generator tool
Likwid tools

DEMO
MICROBENCHMARKING FOR ARCHITECTURAL EXPLORATION

Probing of the memory hierarchy
Saturation effects in cache and memory
Typical OpenMP overheads
LLC performance on Xeon Phi (1 core)
LLC performance on SandyBridge-EP (1 core)
LLC bandwidth scaling Xeon Phi

bandwidth [MiB/s]

core count

no SMT
4-SMT
LLC bandwidth scaling SandyBridge-EP

bandwidth [MiB/s] vs. core count

- no SMT
- 2-SMT
Memory bandwidth saturation on Xeon Phi

![Graph showing memory bandwidth saturation]

- Update (no SMT)
- Update (4-SMT)
- Copy (no SMT)
- Copy (4-SMT)

Bandwidth [MiB/s] vs. Core count
Memory bandwidth saturation on SandyBridge-EP
Thread synchronization overhead on SandyBridge-EP

Barrier overhead in CPU cycles

<table>
<thead>
<tr>
<th></th>
<th>Intel 13.1.0</th>
<th>GCC 4.7.0</th>
<th>GCC 4.6.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Threads</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared L3</td>
<td>384</td>
<td>5242</td>
<td>4616</td>
</tr>
<tr>
<td>SMT threads</td>
<td>2509</td>
<td>3726</td>
<td>3399</td>
</tr>
<tr>
<td>Other socket</td>
<td>1375</td>
<td>5959</td>
<td>4909</td>
</tr>
</tbody>
</table>

GCC not very competitive

Intel compiler

<table>
<thead>
<tr>
<th></th>
<th>Intel 13.1.0</th>
<th>GCC 4.7.0</th>
<th>GCC 4.6.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full domain</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Socket</td>
<td>1497</td>
<td>14546</td>
<td>14418</td>
</tr>
<tr>
<td>Node</td>
<td>3401</td>
<td>34667</td>
<td>29788</td>
</tr>
<tr>
<td>Node +SMT</td>
<td>6881</td>
<td>59038</td>
<td>58898</td>
</tr>
</tbody>
</table>
### Thread synchronization overhead on AMD Interlagos

**Barrier overhead in CPU cycles**

<table>
<thead>
<tr>
<th></th>
<th>Cray 8.03</th>
<th>GCC 4.6.2</th>
<th>PGI 11.8</th>
<th>Intel 12.1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2 Threads</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared L2</td>
<td>258</td>
<td>3995</td>
<td>1503</td>
<td>128623</td>
</tr>
<tr>
<td>Shared L3</td>
<td>698</td>
<td>2853</td>
<td>1076</td>
<td>128611</td>
</tr>
<tr>
<td>Same socket</td>
<td>879</td>
<td>2785</td>
<td>1297</td>
<td>128695</td>
</tr>
<tr>
<td>Other socket</td>
<td>940</td>
<td>2740 / 4222</td>
<td>1284 / 1325</td>
<td>128718</td>
</tr>
</tbody>
</table>

Intel compiler barrier very expensive on Interlagos

OpenMP & Cray compiler

### Full domain

<table>
<thead>
<tr>
<th></th>
<th>Cray 8.03</th>
<th>GCC 4.6.2</th>
<th>PGI 11.8</th>
<th>Intel 12.1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared L3</td>
<td>2272</td>
<td>27916</td>
<td>5981</td>
<td>151939</td>
</tr>
<tr>
<td>Socket</td>
<td>3783</td>
<td>49947</td>
<td>7479</td>
<td>163561</td>
</tr>
<tr>
<td>Node</td>
<td>7663</td>
<td>167646</td>
<td>9526</td>
<td>178892</td>
</tr>
</tbody>
</table>
Thread synchronization overhead on Intel Xeon Phi

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 😊.
Specific issues with Xeon Phi

- **Single core performance** is fragile (in-order execution, instruction pairing, FMA, long latency instructions, wide SIMD)

- Due to the wide SIMD units **latency hiding** is critical but proper **prefetching** is much more difficult to achieve

- Shared access on the **segmented cache** may heavily impact performance

- The optimal operating window is small (small caches, high barrier cost)
But

- Xeon Phi implements many features which will be part of future mainstream processors:
  - 64byte SIMD (Intel Skylake)
  - High degree of parallelism
  - Gather/Scatter
  - FMA

- It allows a glimpse in the future on real hardware
Comparing the performance of different x86 SIMD instruction sets for a medical imaging application

J. Hofmann, J. Treibig, G. Hager, G. Wellein
Motivation

- Builds upon fastrabbit paper
- Port RabbitCT benchmark to Intel Xeon Phi
- Evaluate influence of different instruction sets and their implementation in a microarchitecture on performance
- Targets:
  - SSE, AVX, IvyBridge-EP, Haswell
  - AVX2, Haswell
  - IMCI, Knights Corner

Computer Tomography reconstruction

General motivation: Reconstruction of 3D data from 2D X-ray projection images.

Here: X-ray projections acquired during an intervention.

Reconstruction should be as fast as possible:
- Interventional
- Time resolved (4D) reconstruction

Method: 3D cone beam reconstruction of high resolution C-arm Computer Tomography dataset

Courtesy of J. Hornegger, H. Hofmann
RabbitCT – 3-D reconstruction: open competition

Open platform for performance-comparison of back projection implementations based on a high resolution C-arm CT dataset of a rabbit

Everyone is invited to submit results

Department of Neuroradiology and Pattern Recognition Lab; Univ. Erlangen-Nuremberg

References:
http://www.rabbitct.com/
http://code.google.com/p/rabbitct
Why is RabbitCT interesting?

- Strong scaling problem
- Volume with $512^3$ voxels ($1024^3$ also available)
- 496 projection images (1248x960px) (2.4GB)
- 13 additions, 5 subtractions, 17 multiplications, 3 divides (can be reduced to 1 reciprocal), single precision
- Data volume on volume 496GB, **streaming pattern**
- **Non-deterministic** data access **pattern** on projection images
- Non-trivial arithmetic (profits from FMA)
- On current architectures **instruction throughput limited**

- Demanding target for SIMD vectorization
- Popular optimization target for GPUs
for(int z=0; z<L; z++) {
    for (int y=0; y<L; y++) {
        for (int x=0; x<L; x++) {

            // PART 1
            float wx = 0 + x * MM;
            float wy = 0 + y * MM;
            float wz = 0 + z * MM;
            // convert from WCS to ICS
            // de-homogenize
            float ix = u / w;
            float iy = v / w;
            //integer indices to access projection image
            int iix = (int)ix;
            int iiy = (int)iy;
            // calculate interpolation weights
            float scalex = ix - iix;
            float scaley = iy - iiy;

        }
    }
}
// PART 2
// load four values for bilinear interpolation
float valbl = 0.0f; float valbr = 0.0f;
float valtr = 0.0f; float valtl = 0.0f;
if (iiy>=0 && iiy<width && iix>=0 && iix<height)
    valbl = I[iiy * width + iix];
if (iiy>=0 && iiy<width && iix+1>=0 && iix+1<height)
    valbr = I[iiy * width + iix + 1];
if (iiy+1>=0 && iiy+1<width && iix>=0 && iix<height)
    valtl = I[(iiy + 1) * width + iix];
if (iiy+1>=0 && iiy+1<width && iix+1>=0 && iix+1<height)
    valtr = I[(iiy + 1)* width + iix + 1];

// PART 3
// perform bilinear interpolation
float valb = (1-scalex)*valbl + scalex*valbr;
float valt = (1-scalex)*valtl + scalex*valtr;
float val = (1-scaley)*valb + scaley*valt;
// add weighted results to voxel
VOL[z*L*L+y*L+x] += val / (w * w);

} // x-loop
} // y-loop
} // z-loop

Part 2 +
Part 3
Preliminary Work

- Use reciprocal
- Eliminate if condition with **padding**
- Work reduction with **clipping mask**
- Store back indices to memory for addressing
- Pairwise load of pixel data with shuffles in correct order

- Improvements in present work:
  - Improved clipping mask
  - Better register scheduling
  - More efficient parameter handling

Results in **x1.25** better performance.
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]
    jkknzd k2, ..L100

..L101:

Haswell:

vpcmpeqw ymm7, ymm7, ymm7
vgatherdps ymm15, [rdi + ymm11 * 4], ymm7
# Testbed

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>IvyBridge-EP</th>
<th>Haswell</th>
<th>Knights Corner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Xeon E5-2660 v2</td>
<td>Xeon E3-1240 v3</td>
<td>Xeon Phi 5110P</td>
</tr>
<tr>
<td>Base Clock</td>
<td>2.2 GHz</td>
<td>3.4 GHz</td>
<td>1.053 GHz</td>
</tr>
<tr>
<td>Sockets/Cores/</td>
<td>2/20/40</td>
<td>1/4/8</td>
<td>1/60/240</td>
</tr>
<tr>
<td>SMT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIMD support</td>
<td>SSE(128bit)</td>
<td>AVX2(256bit)</td>
<td>IMCI(512bit)</td>
</tr>
<tr>
<td></td>
<td>AVX(256bit)</td>
<td>FMA3(256bit)</td>
<td></td>
</tr>
<tr>
<td>Vector registers</td>
<td>16</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Memory system</td>
<td>8 ch. DDR3-1866</td>
<td>2 ch. DDR3-1600</td>
<td>16 ch. GDDR5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Node Peak BW</td>
<td>119.4 GB/s</td>
<td>25.6 GB/s</td>
<td>320 GB/s</td>
</tr>
<tr>
<td>Node Update BW</td>
<td>98 GB/s (81%)</td>
<td>23 GB/s (90%)</td>
<td>168 GB/s (53%)</td>
</tr>
</tbody>
</table>
## Gather microbenchmarking results

<table>
<thead>
<tr>
<th></th>
<th>L1 Cache</th>
<th>L2 Cache</th>
<th>Haswell (L1 Cache)</th>
<th>Haswell (L2 Cache)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Instruction</td>
<td>Loop</td>
<td>Instruction</td>
<td>Loop</td>
</tr>
<tr>
<td>16 per CL</td>
<td>9.0</td>
<td>9.0</td>
<td>13.6</td>
<td>13.6</td>
</tr>
<tr>
<td>8 per CL</td>
<td>4.2</td>
<td>8.4</td>
<td>9.4</td>
<td>18.8</td>
</tr>
<tr>
<td>4 per CL</td>
<td>3.7</td>
<td>14.8</td>
<td>9.1</td>
<td>36.4</td>
</tr>
<tr>
<td>2 per CL</td>
<td>2.9</td>
<td>23.2</td>
<td>8.6</td>
<td>68.8</td>
</tr>
<tr>
<td>1 per CL</td>
<td>2.3</td>
<td>36.8</td>
<td>8.1</td>
<td>129.6</td>
</tr>
</tbody>
</table>

**Serialization for loading several items per CL**

**Haswell:**
- Working prefetching
- Microcode solution

**No working prefetching for gather on KNC**
### Instruction code analysis

<table>
<thead>
<tr>
<th></th>
<th>SSE</th>
<th>AVX</th>
<th>AVX2</th>
<th>IMCI</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Part 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Shuffle</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>21</td>
<td>22</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Total</td>
<td>31</td>
<td>33</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td><strong>FMA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Part 2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>18</td>
<td>32</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Shuffle</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>Total</td>
<td>25</td>
<td>36</td>
<td>11</td>
<td>38</td>
</tr>
<tr>
<td><strong>Part 3</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Shuffle</td>
<td>5</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>6</td>
<td>11</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>Total</td>
<td>13</td>
<td>20</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>
## Static analysis summary

<table>
<thead>
<tr>
<th></th>
<th>IvyBridge-EP</th>
<th>Haswell</th>
<th>KNC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SSE</td>
<td>AVX</td>
<td>AVX2</td>
</tr>
<tr>
<td>Voxels/loop</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Instr. SIMD</td>
<td>69</td>
<td>89</td>
<td>41</td>
</tr>
<tr>
<td>Instr. Scalar</td>
<td>57</td>
<td>46</td>
<td>41</td>
</tr>
<tr>
<td>Instr. Count Efficiency</td>
<td>83%</td>
<td>52%</td>
<td>100%</td>
</tr>
<tr>
<td>Runtime Speedup</td>
<td>3.30</td>
<td>4.06</td>
<td>2.61</td>
</tr>
<tr>
<td>Runtime Efficiency</td>
<td>83%</td>
<td>41%</td>
<td>33%</td>
</tr>
</tbody>
</table>

### Key Points
- Direct correlation
- Bad efficiency

Scalar code uses same instruction set capabilities!
- All results with Turbo
- Great benefits from SMT
- AVX code suffers from critical path dependencies

- SSE is fastest w/o SMT
- AVX2 is the slowest

- SMT crucial
- SIMD difference huge
Full Device Results

- Code scales almost perfectly (+90% parallel efficiency on both architectures)
- KNC is faster but far away from leveraging available performance
## Comparison with generated code

<table>
<thead>
<tr>
<th>System</th>
<th>Version</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>IvyBridge-EP</td>
<td>C Code</td>
<td>4.6 AVX</td>
</tr>
<tr>
<td></td>
<td>OpenMP4</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>ISPC</td>
<td>5.4 AVX</td>
</tr>
<tr>
<td></td>
<td>ASM AVX</td>
<td>6.2</td>
</tr>
<tr>
<td>Haswell</td>
<td>C Code</td>
<td>1.3 AVX2</td>
</tr>
<tr>
<td></td>
<td>OpenMP4</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>ISPC</td>
<td>1.7 AVX</td>
</tr>
<tr>
<td></td>
<td>ASM SSE</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>ASM AVX</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td>ASM AVX2</td>
<td>1.6</td>
</tr>
<tr>
<td>KNC</td>
<td>C Code</td>
<td>6.2</td>
</tr>
<tr>
<td></td>
<td>OpenMP4</td>
<td>5.6</td>
</tr>
<tr>
<td></td>
<td>ISPC</td>
<td>6.2</td>
</tr>
<tr>
<td></td>
<td>ASM IMCI</td>
<td>8.5</td>
</tr>
</tbody>
</table>

- ICC 13.1
- ISPC1.5.0
- OpenMP4 not faster than native code
- ISPC fastest with AVX on Haswell !!
- Handtuned vs. C (ISPC):
  - IvyBridge-EP +26% (+13%)
  - Haswell + 32% (+11%)
  - KNC +27% (+27%)
Comparison with GPU

- Thumper code (GTX680) **8 times faster** than KNC

Reasons:

1. Bilinear interpolation (including the pixel data loading) is implemented completely in hardware (**texture units**). 90% of kernel time on KNC are spent on this. On GPU this is one instruction.

2. More robust **latency hiding** mechanisms. Gather instruction latencies and latencies of non contiguous data access are problematic on KNC.
Conclusion

- Employing wider SIMD units is **not efficient** for this algorithm with its non-contiguous data access

- The new **gather** instructions allow for a **simpler code** but do not improve on performance

- **Code generators** are more competitive on the gather enabled architectures

- GPUs are faster because they provide **application specific** hardware

- KNC has a lot of shortcomings