Vectorization
Why you shouldn’t care.

June 7th, 2016  |  Ivo Kabadshow, Andreas Beckmann  |  Jureca Workshop 2016, JSC
Welcome To The Lectures On Hardware Design

- Processor microarchitecture
- Micro-Ops bandwidth/latency
- Memory hierarchy
- Memory bandwidth/latency
- Critical path analysis
- Intrinsics/Assembly

Instead we only look at this more conceptually.

June 7th, 2016
Ivo Kabadshow, Andreas Beckmann
Welcome To The Lectures On Hardware Design

- Processor microarchitecture
- Micro-Ops bandwidth/latency
- Memory hierarchy
- Memory bandwidth/latency
- Critical path analysis
- Intrinsics/Assembly

Instead we only look at this more conceptually.
Why You Should Not Care A Lot

You have to know a lot about your code

- data layout, algorithmic complexity, critical path, performance bottlenecks

You need to change a lot

- algorithm, data layout, loop structure, access pattern

You probably make it worse

- non-portable binary, confused compiler, unreadable and bloated source code
What is your ...  
Or: Is vectorization the right tool for me?

... performance bottleneck?
- FLOPS, memory (bandwidth/latency)
- Data dependencies
- Communication (MPI), I/O (Disk)

... programming language?
- Fortran – Are you allowed/capable to write ASM,C/C++ code?
- Python – Are you allowed/capable to write ASM,C/C++ code?
- C/C++ – Already close to the metal, go ahead ...
### Level of Parallelism

#### Parallelization Stack

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td></td>
</tr>
<tr>
<td>User</td>
<td>Compiler extensions (OMP)</td>
</tr>
<tr>
<td>Compiler, User</td>
<td></td>
</tr>
<tr>
<td>Compiler, Hardware</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td></td>
</tr>
</tbody>
</table>

#### Vectorization

- Fine-grained parallelism needs regular algorithmic structure
Level of Parallelism

Parallelization Stack

<table>
<thead>
<tr>
<th>Parallelization</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>User</td>
</tr>
<tr>
<td>Threading</td>
<td>User, Compiler extensions (OMP)</td>
</tr>
<tr>
<td>Cache Tiling</td>
<td>Compiler, User</td>
</tr>
<tr>
<td>Unrolling</td>
<td>Compiler, Hardware</td>
</tr>
<tr>
<td>Vectorization</td>
<td>Compiler</td>
</tr>
</tbody>
</table>

**Vectorization**

- Fine-grained parallelism needs regular algorithmic structure

**Best practices: Find Critical Kernel Algorithm**

- Use highly tuned existing libraries, like: Intel MKL, Atlas, BLAS, Eigen, FFTW, Spiral, ...
Memory Hierarchy

Data Movement Is Very Expensive

- Towards top: more expensive ($), smaller, faster
- Reuse data in registers, cache as much as possible
Microarchitecture
Memory, Caches, Register, Ports
Microarchitecture
Memory, Caches, Register, Ports
Single Instruction - Multiple Data (SIMD)

### SIMD Operations in AVX

<table>
<thead>
<tr>
<th>Float/Int:</th>
<th>+</th>
<th>−</th>
<th>×</th>
<th>÷</th>
<th>√ ·</th>
<th></th>
<th>min(⋅)</th>
<th>max(⋅)</th>
<th>sgn(⋅)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double/Long:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Some Available Operations

- +, −, ×, ÷, √, | ⋅ | min(⋅), max(⋅), sgn(⋅)

### Careful!

- Consecutive (and aligned) memory for SIMD load/store needed
- Not all operations are available in SIMD (trigonometric, exp, log)
- Some operations may be available as estimates (1/⋅, √⋅)
- Not all operations have the same ALU bandwidth or latency
Using Intrinsics to Utilize SIMD
C/C++ Intrinsics, Use up to 16 registers

1/√a estimate with 22 bits for 8 floats

```c
inline __m256 rsqrt_estimate_22bits(__m256 a)
{
    __m256 ymm0;
    __m256 ymm2;
    __m256 ymm3 = {3.f, 3.f, 3.f, 3.f, 3.f, 3.f, 3.f, 3.f};
    __m256 ymm4 = {0.5f, 0.5f, 0.5f, 0.5f, 0.5f, 0.5f, 0.5f, 0.5f};
    __m256 ymm5;
    ymm5 = a;
    ymm0 = _mm256_rsqrt_ps(ymm5);
    ymm2 = ymm0 * ymm0;
    ymm2 = ymm2 * ymm5;
    ymm2 = ymm3 - ymm2;
    ymm0 = ymm0 * ymm2;
    ymm0 = ymm0 * ymm4;
    return ymm0;
}
```
Single Instruction - Multiple Data (SIMD)

Fused Multiply-Add (FMA) – Three Operands (FMA3)

- No SIMD (float or double)

\[ = \times + \]

- SIMD (Double)

\[ = \times + \]

- SIMD (Float)

\[ = \times + \]

Details

- One operand may come from memory
- One operand needs to reused for output (\(=\))
## Perfect ASM for OoOE on Intel CPUs

Compiler-generated from SIMD-aware code

### AVX without FMA

```assembly
loop:
  vsubps 0x0(%r13,%rax,4),%ymm12,%ymm7
  vsubps (%r14,%rax,4),%ymm13,%ymm6
  vsubps (%r15,%rax,4),%ymm14,%ymm5
  vmulps %ymm7,%ymm7,%ymm8
  vmulps %ymm6,%ymm6,%ymm0
  vaddps %ymm0,%ymm8,%ymm0
  vsqrtps %ymm0,%ymm0
  vdivps %ymm0,%ymm9,%ymm0
  vmulps (%rdu,%rax,4),%ymm8,%ymm8
  add $0x8,%rax
  cmp %rax,%rbx
  vmulps %ymm0,%ymm0,%ymm0
  vmulps %ymm0,%ymm8,%ymm0
  vaddps %ymm0,%ymm8,%ymm0
  vaddps %ymm9,%ymm9,%ymm9
  vaddps %ymm0,%ymm4,%ymm4
  ja  loop
```

### AVX2 with FMA

```assembly
loop:
  vsubpd (%rcx,%rdx,8),%ymm13,%ymm7
  vsubpd (%rdi,%rdx,8),%ymm14,%ymm6
  vsubpd (%r8,%rdx,8),%ymm15,%ymm5
  vmulpd %ymm6,%ymm6,%ymm4
  vfmadd231pd %ymm7,%ymm7,%ymm4
  vfmadd231pd %ymm5,%ymm5,%ymm4
  vsqrtpd %ymm4,%ymm4
  vdivpd %ymm4,%ymm12,%ymm4
  vmulpd (%r9,%rdx,8),%ymm4,%ymm8
  add $0x4,%rdx
  vfmadd231pd %ymm4,%ymm7,%ymm0
  vfmadd231pd %ymm4,%ymm6,%ymm1
  vfmadd231pd %ymm4,%ymm5,%ymm2
  jb  loop
```
Demystifying Peak-Performance

<table>
<thead>
<tr>
<th>Jureca Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>1872 compute nodes</td>
</tr>
<tr>
<td>Two Intel Xeon E5-2680 v3 Haswell CPUs per node</td>
</tr>
<tr>
<td>2 x 12 cores, 2.5 GHz, SMT</td>
</tr>
<tr>
<td>AVX 2.0 ISA extension</td>
</tr>
<tr>
<td>1.8 PFLOPs for all the CPUs combined</td>
</tr>
</tbody>
</table>
Demystifying Peak-Performance

Jureca Specifications

- 1872 compute nodes
- Two Intel Xeon E5-2680 v3 Haswell CPUs per node
- 2 x 12 cores, 2.5 GHz, SMT
- AVX 2.0 ISA extension
- 1.8 PFLOPs for all the CPUs combined

What is the performance of your code on Jureca?
Dissecting Peak-Performance
Processor Architecture: Intel Haswell, Broadwell & Skylake

<table>
<thead>
<tr>
<th>FLOP/Cycle</th>
<th>Single Element (4/8 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>• Float: 32 (256 bit SIMD size: 8 elements × 4 Bytes)</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{Float} & = \text{Float} \times \text{Float} + \text{Float} \\
\text{Float} & = \text{Float} \times \text{Float} + \text{Float}
\end{align*}
\]

| • Double: 16 (256 bit SIMD size: 4 elements × 8 Bytes) |

\[
\begin{align*}
\text{Double} & = \text{Double} \times \text{Double} + \text{Double} \\
\text{Double} & = \text{Double} \times \text{Double} + \text{Double}
\end{align*}
\]

- Throughput: 2 per cycle (super-scalar), 2 FMA ports available
- Latency: 5 cycles, pipelining needed for peak performance
### Dissecting Peak-Performance

#### Peak Performance

- 32 FLOP/cycle $\times$ 2.5GHz $\times$ 12 cores $\times$ 2 sockets $\times$ 1872 nodes
  
  $= 3.5942 \times 10^{15}$ (float)

  $= 1.7971 \times 10^{15}$ (double) with 16 FLOP/cycle

---

- Pipelined instructions
- Only 1 store and 2 loads per cycle possible
- Data resides in L1 cache, no cache misses
- No data dependencies
- No I/O, no MPI bottleneck
WTF?

Realistic FLOPs Divider

Double Precision: 2
No FMA used: 2
One FP port per cycle: 2
No SIMD: 4

1/32 \approx 3\% peak is very good!
### Realistic FLOPs vs. Divider

<table>
<thead>
<tr>
<th></th>
<th>divider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double Precision</td>
<td>2</td>
</tr>
<tr>
<td>No FMA used</td>
<td>2</td>
</tr>
<tr>
<td>One FP port per cycle</td>
<td>2</td>
</tr>
<tr>
<td>No SIMD</td>
<td>4</td>
</tr>
</tbody>
</table>

- $\frac{1}{32} \approx 3\%$ peak is very good!
Data Layout vs. SIMDization

Your data structures need to adapt

- **Array of Structs (AoS)**
  - xyzxyzxyzxyz, …

- **Struct of Arrays (SoA)**
  - xxxx …
  - yyyy …
  - zzzz …

- **Array of Struct of Arrays (AoSoA)**
  - xxyyzzxxyyzz …

**SIMDization**

- In which direction does the operation go?
- Do you access your data aligned (32 Byte)?
Pitfalls I
If you still decide to do it yourself

- Data alignment is paramount
- Reordering/shuffling of data might increase runtime

- Algorithm needs to be adapted to hardware architecture
- Algorithm will be most likely not portable

- Expert knowledge of microarchitecture required
- Expert knowledge of instruction set architecture (ISA) required
Pitfalls II

- Use of SIMD may reduce clockspeed
- Some SIMD instructions are not fully vectorized (√·)
- Increased register usage may be a bottleneck

- Increased performance after redesign may come from CPU out-of-order unit
- Data dependencies on critical path may stall computation

- Don’t optimize for a specific SIMD size
- SIMD size will increase in future (e.g. Xeon Phi)
Going to utilize a GPU soon?

- Listen carefully to the GPU talk
- Start porting to the GPU first
- Algorithm has to change anyway
- Prepare loop unrolling, find common/independent sub-expressions
- Optimized GPU code (SIMT) can be ported back to CPU (SIMD)

Lessons learned

- Solution depends on your problem/bottleneck
- If you are paid for science, don’t go off-course too much
- Change data layout to allow the use of libraries
- Ask you boss to employ a computer scientist
I am using SIMDized libraries and do not care anymore!
Vectorization
Why you shouldn’t care.

June 7th, 2016 | Ivo Kabadshow, Andreas Beckmann | Jureca Workshop 2016, JSC