From Zero To Hero
Understanding and fixing on-core performance bottlenecks

Course Material

Examples
All examples can be found here:
https://gitlab.version.fz-juelich.de/beckmann1/FromZeroToHero2019.git
Part I: Basics

HPC Today

Lots of Opportunity To Improve
- New, fancy hardware to play with
- FLOPs are very cheap

Lots of Opportunity To Mess Up
- What works well on A, might not run at all on B
- Peak-performance is limited to a few algorithms
Maintainability vs Performance

Challenges
Goal: Optimal time to solution on every platform

- **Flexibility**
  - Hardware: CPU/GPU
  - ILP, SIMD, OoOE
  - Cache levels & sizes
  - NUMA
  - Threading & MPI

- **Configurability**
  - Algorithm
  - Different implementations
  - Different critical paths

- **Customization**
  - Application
  - Physical model
  - Accuracy range
  - System size
Moore’s Law
The free lunch is over. Now we have more cores and complexer cores

Top500 List – Historic Data
Who is responsible for performance?

- Developer
- Compiler
- Hardware

Where is my performance?

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
Flynn Taxonomy

Processor Organization

SISD
- uni processor
SIMD
- vector processor
- array processor
MISD
MIMD
- shared memory
- distributed memory
- UMA
- NUMA
- MPI

Jureca Hardware

One node of Jureca
- Vectorization within each core, threading on a socket, MPI between sockets or nodes
Memory Hierarchy

- Modern hardware consists of a hierarchy of memory
- The closer to the core (L1), the faster + more expensive
- Can not be addressed directly
- Locality has huge impact on performance
- If you load something into the CPU, do as much as possible with the data
Memory Bandwidth
Full Memory Hierarchy: Intel Skylake

Bandwidth [B/cycle]
Array Size $\log_2 [B]$

- Red dots: Read – Scan512UnrollLoop
- Black squares: Write – Scan512UnrollLoop

Bandwidth in GByte/s

Part II: A Simple Reduction Example
Example 1: Simple Reduction

\[ A = \sum_{i=0}^{n} a_i \]

- numerical round-off errors?
- binary reproducibility?
- reordering allowed?

- the compiler does not know!

Goodbye Constant Peak-Performance

Turbo modes for Xeon E5-2680 v3 (Haswell)
Simple Reduction – Code Example

Basic Implementation

```cpp
Real naive_sum(const Vector &v, size_t n) {
    n = std::min(n, v.size());
    Real s = 0.0;
    for (size_t i = 0; i < n; ++i)
        s += v[i];
    return s;
}
```

```
cd ./examples/reduce/000-un-throttling
make
./reduce-float
./reduce-double
```
Pitfall

- Modern processors change frequency depending on thermal envelope
- Turbo-Boost can be disabled on some CPUs
- There still might be other hardware features changing the clock speed
- ✔ If you want to measure fine-grained performance, make sure to pre-heat with compute load

Simple Reduction – Code Example

Basic Implementation

```cpp
Real naive_sum(const Vector &v, size_t n)
{
    n = std::min(n, v.size());
    Real s = 0.0;
    for (size_t i = 0; i < n; ++i)
        s += v[i];
    return s;
}
```

- cd ./examples/reduce/001-small-sizes/
- make
- ./reduce-float
- ./reduce-double
Simple reduction – Output
Measuring only a few instructions is difficult

```bash
train000@zam1234:~/examples/reduce/001-small-sizes$ ./reduce-float
heating up cpu and caches...
naive : n=1000 sum=-3.8388587952e+01 time= 0.001 ms
naive : n=1000 sum=-3.8388587952e+01 time= 0.001 ms
...  
naive : n=1000 sum=-3.8388587952e+01 time= 0.001 ms
naive : n=1000 sum=-3.8388587952e+01 time= 0.001 ms
naive: ( 1 Op = 'accumulator += v[i]')

n= 1000 ( 4000 bytes) time= 0.000891 ms GOp/s= 1.12233 INS/Op= 4.417 CYC/Op= 4.929
n= 1000 ( 4000 bytes) time= 0.000849 ms GOp/s= 1.17786 INS/Op= 4.417 CYC/Op= 4.178
n= 1000 ( 4000 bytes) time= 0.000850 ms GOp/s= 1.17647 INS/Op= 4.417 CYC/Op= 4.033
n= 1000 ( 4000 bytes) time= 0.001144 ms GOp/s= 0.87413 INS/Op= 4.417 CYC/Op= 5.139
```

Pitfall
- Measuring only a few instructions gives wrong results
- The measuring overhead will be included in the measurement
- The time resolution itself is limited
- Increase the repetition factor to get stable measurements
Simple Reduction – Code Example

Basic Implementation

```cpp
Real naive_sum(const Vector &v, size_t n)
{
    n = std::min(n, v.size());
    Real s = 0.0;
    for (size_t i = 0; i < n; ++i)
        s += v[i];
    return s;
}
```

```
$ cd /examples/reduce/01-naive
$ make
$ ./reduce-float
$ ./reduce-double
```

Simple reduction – Output

```
train000@zam1234:$ cd /examples/reduce/01-naive
train000@zam1234:~/examples/reduce/01-naive$ ./reduce-float
heating up cpu and caches...
naive : n=2000 sum=2.4007868652e+03 time= 16.654 ms
naive : n=2000 sum=2.4007868652e+03 time= 17.000 ms
... 
naive : n=2000 sum=2.4007868652e+03 time= 16.512 ms
naive: (1 Op 'accumulator += v[i]')
n= 2000 ( 8000 bytes) time= 16.812190 ms GOp/s= 1.18961 INS/Op= 4.000 CYC/Op= 3.113
n= 2000 ( 8000 bytes) time= 17.667315 ms GOp/s= 1.13203 INS/Op= 4.000 CYC/Op= 3.272
```
Can this baseline be improved?

- Is there a performance difference between float and double?
- Is there a performance difference for different size?
- What are the limiting factors?
- What can we do to remove the limit?
- What is your guess? How much faster can we get?

We need a better understanding of the hardware!

---

Jureca Hardware

One node of Jureca

- Vectorization within each core, threading on a socket, MPI between sockets or nodes

![Jureca Hardware Diagram]
How To Compute 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

Where does the 1.8 PFLOPs peak performance come from?
How To Compute Peak Performance
Processor Architecture: Intel Haswell, Broadwell & Skylake

FLOPs/Cycle Single Element (4/8 Bytes)

- **Float**: 32 (256 bit SIMD size: 8 elements × 4 Bytes)
  \[ \text{FLOPs} = \text{elements} \times \text{bytes} \times \text{cycles} \]

- **Double**: 16 (256 bit SIMD size: 4 elements × 8 Bytes)
  \[ \text{FLOPs} = \text{elements} \times \text{bytes} \times \text{cycles} \]

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

---

How To Compute Peak Performance

Peak Performance

- 32 FLOP/cycle × 2.5GHz × 12 cores × 2 sockets × 1872 nodes
  \[ \approx 3.5942 \times 10^{15} \approx 3.6 \text{PFLOPs (float)} \]

- 1.7971 × 10^{15} \approx 1.8 \text{PFLOPs (double)} \quad \text{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
Intel Processor Generations

2010 2012 2014 2016 2018
Sandybridge 22nm Ivybridge
AVX256 32nm
Ivybridge
AVX256 32nm
Haswell
FMA
Broadwell
14nm
Skylake
14nm
Kabylake
14nm
Skylake-SP
AVX512

Jureca Microarchitecture: Haswell (simplified)
Out-of-Order Unit can Reshuffle Independent Ops

Instructions
Decoder
Out of Order Unit
ALU ALU ALU Load Load Store Branch
Mul Add Shuffle Jump
Div
FMA FMA
Vec Vec Vec Data Data Data
Part III: Let’s Optimize the Code

Simple Reduction – Code Example

Basic Implementation

```cpp
Real naive_sum(const Vector &v, size_t n) {
    n = std::min(n, v.size());
    Real s = 0.0;
    for (size_t i = 0; i < n; ++i)
        s += v[i];
    return s;
}
```

cd /examples/reduce/01-naive
make
./reduce-float
./reduce-double
Simple reduction – Output

```
train000@zam1234:~/examples/reduce/01-naive$ ./reduce-float
heating up cpu and caches...
naive : n=2000 sum=2.4007868652e+03 time= 16.654 ms
naive : n=2000 sum=2.4007868652e+03 time= 17.000 ms
...
naive : n=2000 sum=2.4007868652e+03 time= 16.512 ms

naive: (1 Op 'accumulator += v[i]')
n= 2000 ( 8000 bytes) time= 16.812190 ms GOp/s= 1.18961 INS/Op= 4.000 CYC/Op= 3.113
n= 2000 ( 8000 bytes) time= 17.667315 ms GOp/s= 1.13203 INS/Op= 4.000 CYC/Op= 3.272
```
Real interleaved_sum(const Vector &v, size_t n) {
    const size_t displacement = 64 / sizeof(Real);
    n = std::min(n, v.size());
    Real s = 0.0;
    for (size_t k = 0; k < displacement; ++k)
        for (size_t i = k; i < n; i += displacement)
            s += v[i];
    return s;
}
Memory Bandwidth
Full Memory Hierarchy: Jureca 256bit load/store

```
Read – Scan256UnrollLoop
Write – Scan256UnrollLoop
Read – Scan64UnrollLoop
Write – Scan64UnrollLoop
```

Simple Reduction – Code Example

```
Real naive_sum(const Vector &v, size_t n)
{
    n = std::min(n, v.size());
    Real s = 0.0;
    for (size_t i = 0; i < n; ++i)
        s += v[i];
    return s;
}
```

```
cd /examples/reduce/01-naive
make
./reduce-float
./reduce-double
```
Simple reduction – Output

```
train000@zam1234:~/examples/reduce/01-naive$ ./reduce-float
heating up cpu and caches...
naive : n=2000 sum=2.4007868652e+03 time= 16.654 ms
naive : n=2000 sum=2.4007868652e+03 time= 17.000 ms
...  
naive : n=2000 sum=2.4007868652e+03 time= 16.512 ms

naive: (1 Op 'accumulator += v[i]')
n= 2000 ( 8000 bytes) time= 16.812190 ms GOp/s= 1.18961 INS/Op= 4.000 CYC/Op= 3.113
n= 2000 ( 8000 bytes) time= 17.667315 ms GOp/s= 1.13203 INS/Op= 4.000 CYC/Op= 3.272
```
Simple Reduction – Code Example

Unrolling 2×

```cpp
Real naive_unroll2_sum(const Vector &v, size_t n)
{
    n = std::min(n, v.size());
    Real s = 0.0;

    size_t i = 0;
    for (; i + 2 <= n; i += 2)
    {
        s += v[i + 0];
        s += v[i + 1];
    }
    for (; i < n; ++i)
        s += v[i];

    return s;
}
```

Instruction Scheduling

<table>
<thead>
<tr>
<th></th>
<th>Load</th>
<th>Add</th>
<th>Increment</th>
<th>Compare</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instruction Latency and Throughput (SIMD+Type)
On Intel Haswell

### Latency Costs: ADD/MUL

<table>
<thead>
<tr>
<th>SIMD</th>
<th>float</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>3/5</td>
<td>3/5</td>
</tr>
<tr>
<td>128</td>
<td>3/5</td>
<td>3/5</td>
</tr>
<tr>
<td>256</td>
<td>3/5</td>
<td>3/5</td>
</tr>
</tbody>
</table>

### Reciprocal Throughput: ADD, MUL

<table>
<thead>
<tr>
<th>SIMD</th>
<th>float</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Latency Costs: SQRT, DIV

<table>
<thead>
<tr>
<th>SIMD</th>
<th>float</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>128</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>256</td>
<td>19</td>
<td>29</td>
</tr>
</tbody>
</table>

### Reciprocal Throughput: SQRT, DIV

<table>
<thead>
<tr>
<th>SIMD</th>
<th>float</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>256</td>
<td>14</td>
<td>28</td>
</tr>
</tbody>
</table>
**Instruction Latency and Throughput (SIMD+Type)**
On Intel Haswell

**Latency Costs: RSQRTE**

<table>
<thead>
<tr>
<th>SIMD</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>5</td>
</tr>
<tr>
<td>128</td>
<td>5</td>
</tr>
<tr>
<td>256</td>
<td>7</td>
</tr>
</tbody>
</table>

**Reciprocal Throughput: RSQRTE**

<table>
<thead>
<tr>
<th>SIMD</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>2</td>
</tr>
</tbody>
</table>

- RSQRTE only has a precision of 11 bit, float=23 bit, double=52 bit
- Newton-Raphson to increase precision possible subsequently
- AVX512 has 14 bit version called RSQRT14
- Intel KNL has 28 bit version called RSQRT28

---

**Simple Reduction – Code Example**

**Unrolling 2× with Extra Accumulator**

```c
Real unroll2_sum(const Vector &v, size_t n)
{
    n = std::min(n, v.size());
    Real s0 = 0.0, s1 = 0.0;

    size_t i = 0;
    for ( ; i + 2 <= n; i += 2)
    {
        s0 += v[i + 0];
        s1 += v[i + 1];
    }
    Real s = s0 + s1;
    for (; i < n; ++i)
        s += v[i];

    return s;
}
```

---

Member of the Helmholtz Association May 16, 2019 Slide 42

Member of the Helmholtz Association May 16, 2019 Slide 43
Simple Reduction – Code Example
Unrolling 3× with Extra Accumulator

```cpp
Real unroll3_sum(const Vector &v, size_t n)
{
    n = std::min(n, v.size());
    Real s0 = 0.0, s1 = 0.0, s2 = 0.0;
    size_t i = 0;
    for ( ; i + 3 <= n; i += 3)
    {
        s0 += v[i + 0];
        s1 += v[i + 1];
        s2 += v[i + 2];
    }
    Real s = s0 + s1 + s2;
    for (; i < n; ++i)
        s += v[i];
    return s;
}
```

Instruction Scheduling

![Instruction Scheduling Diagram]

- Load
- Add
- Increment
- Compare
- Branch
Simple Reduction – Code Example
Unrolling $n \times$ with Unroll Loop

```cpp
Real dynamic_unroll3_sum(const Vector &v, size_t n)
{
    n = std::min(n, v.size());
    const size_t unroll = runtime_constant(size_t(3));
    Real su[unroll];
    for (auto &r: su)
        r = 0.0;

    size_t i = 0;
    for ( ; i + unroll <= n; i += unroll)
        for (size_t j = 0; j < unroll; ++j)
            su[j] += v[i + j];
    Real s = 0.0;
    for (i < n; ++i)
        s += v[i];

    for (const auto &r: su)
        s += r;

    return s;
}
```

Simple Reduction – Code Example
Unrolling 4\times with Extra Accumulator

```cpp
Real unroll4_sum(const Vector &v, size_t n)
{
    n = std::min(n, v.size());
    Real s0 = 0.0, s1 = 0.0, s2 = 0.0, s3 = 0.0;

    size_t i = 0;
    for ( ; i + 4 <= n; i += 4)
    {
        s0 += v[i + 0];
        s1 += v[i + 1];
        s2 += v[i + 2];
        s3 += v[i + 3];
    }
    Real s = s0 + s1 + s2 + s3;
    for ( ; i < n; ++i)
        s += v[i];

    return s;
}
```
Simple Reduction – Code Example
Unrolling 5× with Extra Accumulator

```cpp
Real unroll5_sum(const Vector &v, size_t n)
{
    n = std::min(n, v.size());
    Real s0 = 0.0, s1 = 0.0, s2 = 0.0, s3 = 0.0, s4 = 0.0;

    size_t i = 0;
    for (; i + 5 <= n; i += 5)
    {
        s0 += v[i + 0];
        s1 += v[i + 1];
        s2 += v[i + 2];
        s3 += v[i + 3];
        s4 += v[i + 4];
    }
    Real s = s0 + s1 + s2 + s3 + s4;
    for (; i < n; ++i)
        s += v[i];

    return s;
}
```

Simple Reduction – Code Example
Unrolling 16× with Extra Accumulator

```cpp
Real unroll16_sum(const Vector &v, size_t n)
{
    n = std::min(n, v.size());
    Real s0 = 0.0, s1 = 0.0, ... , s14 = 0.0, s15 = 0.0;

    size_t i = 0;
    for (; i + 16 <= n; i += 16)
    {
        s0 += v[i + 0];
        s1 += v[i + 1];
        // ...
        s14 += v[i + 14];
        s15 += v[i + 15];
    }
    Real s = s0 + s1 + ... + s14 + s15;
    for (; i < n; ++i) s += v[i];

    return s;
}
```
**Simple Reduction – Code Example**

*Unrolling 17× with Extra Accumulator*

```cpp
Real unroll17_sum(const Vector &v, size_t n)
{
    n = std::min(n, v.size());
    Real s0 = 0.0, s1 = 0.0, ... , s15 = 0.0, s16 = 0.0;

    size_t i = 0;
    for (; i + 17 <= n; i += 17)
    {
        s0 += v[i + 0];
        s1 += v[i + 1];
        // ...
        s15 += v[i + 15];
        s16 += v[i + 16];
    }
    Real s = s0 + s1 + ... + s15 + s16;
    for (; i < n; ++i) s += v[i];
    return s;
}
```

---

**Instruction Level Parallelism (ILP)**

- We saturate port 1: Add ALU
- There are no more add ALUs available
- Every cycle is used to issue a floating-point add instruction

- Are we utilizing all available units for A+=a[i]?
- Can we increase port occupancy?
Simple Reduction – Code Example

Unrolling 2x with Extra Accumulator and FMA

```cpp
Real fma_unroll2_sum(const Vector &v, size_t n)
{
    n = std::min(n, v.size());
    Real scale = runtime_constant(1.0);
    Real s0 = 0.0, s1 = 0.0;

    size_t i = 0;
    for (; i + 2 <= n; i += 2)
    {
        s0 += v[i + 0] * scale;
        s1 += v[i + 1] * scale;
    }
    Real s = s0 + s1;
    for (; i < n; ++i)
        s += v[i] * scale;
    return s;
}
```

Unrolling 5x with Extra Accumulator and FMA

```cpp
Real fma_unroll5_sum(const Vector &v, size_t n)
{
    n = std::min(n, v.size());
    Real scale = runtime_constant(1.0);
    Real s0 = 0.0, s1 = 0.0, s2 = 0.0, s3 = 0.0, s4 = 0.0;

    size_t i = 0;
    for (; i + 5 <= n; i += 5)
    {
        s0 += v[i + 0] * scale;
        s1 += v[i + 1] * scale;
        s2 += v[i + 2] * scale;
        s3 += v[i + 3] * scale;
        s4 += v[i + 4] * scale;
    }
    Real s = ((s0 + s1) + (s2 + s3)) + s4;
    for (; i < n; ++i)
        s += v[i] * scale;
    return s;
}
```


```cpp
Real fma_unroll10_sum(const Vector &v, size_t n)
{
    n = std::min(n, v.size());
    Real scale = runtime_constant(1.0);
    Real s0 = 0.0, s1 = 0.0, ... , s9 = 0.0;

    size_t i = 0;
    for ( ; i + 10 <= n; i += 10)
    {
        s0 += v[i + 0] * scale;
        s1 += v[i + 1] * scale;
        // ...
        s9 += v[i + 9] * scale;
    }
    Real s = ((s0 + s1) + ... + (s8 + s9));

    for (; i < n; ++i)
        s += v[i] * scale;

    return s;
}
```

**Part V: Let’s SIMDize the Code**
Single Instruction – Multiple Data (SIMD)

Classical vs SIMD Operations in AVX

| Double/Long: | ⊙ | ⊙ |
| Float/Int:   | ⊙ | ⊙ |

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

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

- No SIMD (float or double)
  
  - SIMD (Double)

  - SIMD (Float)

Details

- One operand may come from memory
- One register operand needs to be reused for output (=)
What kind of vectorization is supported?

- cat /proc/cpuinfo
- cat /proc/cpuinfo | grep 'model name'
- cat /proc/cpuinfo | grep 'processor'
- cat /proc/cpuinfo | grep 'cpu MHz'
- cat /proc/cpuinfo | grep 'flags'

Example for float data type

- scalar
- SSE
- AVX
- AVX2
- AVX512

requirement: memory should be aligned
operation on misaligned data on some platforms not allowed
on others allowed but significantly slower
Peel – Aligned Load, Repeat – Tail
Make sure to only access aligned data

Simple Reduction – Code Example
Using AVX256 Vectorization with alignment

```c++
Real simd_sum(const Vector &v, size_t n)
{
    const size_t simd_width = sizeof(SimdReal) / sizeof(Real);
    n = std::min(n, v.size());
    Real s = 0.0;
    size_t i = 0;
    // scalar ops: peel until the pointer is aligned
    for (; i < n && ((reinterpret_cast<intptr_t>(&v[i]) % 64) != 0); ++i)
        s += v[i];
    // SIMD ops
    SimdReal simd_acc = { 0.0 };  
    for (; i + simd_width <= n; i += simd_width)
        simd_acc += reinterpret_cast<const SimdReal*>(&v[i]);
    for (size_t j = 0; j < simd_width; ++j)
        s += reinterpret_cast<const Real*>(&simd_acc)[j];
    // scalar ops: tail
    for (; i < n; ++i) s += v[i];
    return s;
}
```
Unaligned Load, Repeat – Tail
Load unaligned, but assume aligned

no peeling
unaligned SIMD load 1,2,3,4,5
scalar tail load 6

Simple Reduction – Code Example
Using AVX256 Vectorization, but misaligned memory

```cpp
Real misaligned_simd_sum(const Vector &v, size_t n) {
    const size_t misalign = 1 * sizeof(Real);
    const size_t simd_width = sizeof(SimdReal) / sizeof(Real);
    n = std::min(n, v.size());
    Real s = 0.0;
    size_t i = 0;
    for ( ; i < n && ((reinterpret_cast<intptr_t>(&v[i]) % 64) != misalign); ++i)
        s += v[i];
    SimdReal simd_acc = { 0.0 };
    for (; i + simd_width <= n; i += simd_width)
        simd_acc += *(reinterpret_cast<const SimdReal*>(&v[i]));
    for (size_t j = 0; j < simd_width; ++j)
        s += reinterpret_cast<const Real*>(&simd_acc)[j];
    return s;
}
```
Simple Reduction – Code Example
Using AVX256 Vectorization, alignment and unrolling 2x

Real simd_unroll2_sum(const Vector &v, size_t n) {
    const size_t simd_width = sizeof(SimdReal) / sizeof(Real);
    n = std::min(n, v.size());
    Real s = 0.0;
    size_t i = 0;
    for ( ; i < n && ((reinterpret_cast<intptr_t>(&v[i]) % 64) != 0); ++i) s += v[i];
    SimdReal simd_acc0 = { 0.0 }; SimdReal simd_acc1 = { 0.0 };
    for (; i + 2 * simd_width <= n; i += 2 * simd_width) {
        simd_acc0 += *reinterpret_cast<const SimdReal*>(&v[i]);
        simd_acc1 += *reinterpret_cast<const SimdReal*>(&v[i + simd_width]);
    }
    for ( ; i < n; ++i) s += v[i];
    simd_acc0 += simd_acc1;
    for (size_t j = 0; j < simd_width; ++j)
        s += reinterpret_cast<const Real*>(&simd_acc0)[j];
    return s;
}

Simple Reduction – Code Example
Using AVX256 Vectorization, alignment and unrolling 3x

Real simd_unroll3_sum(const Vector &v, size_t n) {
    const size_t simd_width = sizeof(SimdReal) / sizeof(Real);
    n = std::min(n, v.size());
    Real s = 0.0;
    size_t i = 0;
    for ( ; i < n && ((reinterpret_cast<intptr_t>(&v[i]) % 64) != 0); ++i) s += v[i];
    SimdReal simd_acc0 = { 0.0 }; SimdReal simd_acc1 = { 0.0 }; SimdReal simd_acc2 = { 0.0 };
    for (; i + 3 * simd_width <= n; i += 3 * simd_width) {
        simd_acc0 += *reinterpret_cast<const SimdReal*>(&v[i + 0 * simd_width]);
        simd_acc1 += *reinterpret_cast<const SimdReal*>(&v[i + 1 * simd_width]);
        simd_acc2 += *reinterpret_cast<const SimdReal*>(&v[i + 2 * simd_width]);
    }
    for ( ; i < n; ++i) s += v[i];
    simd_acc0 += simd_acc1 + simd_acc2;
    for (size_t j = 0; j < simd_width; ++j) s += reinterpret_cast<const Real*>(&simd_acc0)[j];
    return s;
}
Simple Reduction – Code Example

Using AVX256 Vectorization, misalignment and unrolling $3 \times$

```cpp
Real misaligned_simd_unroll3_sum(const Vector &v, size_t n)
{
    const size_t misalign = 1 * sizeof(Real);
    const size_t simd_width = sizeof(SimdReal) / sizeof(Real);
    n = std::min(n, v.size());
    Real s = 0.0;
    size_t i = 0;
    for (; i < n && ((reinterpret_cast<intptr_t>(&v[i]) % 64) != misalign); ++i) s += v[i];
    SimdReal simd_acc0 = { 0.0 };  
    SimdReal simd_acc1 = { 0.0 };  
    SimdReal simd_acc2 = { 0.0 };  
    for (; i + 3 * simd_width <= n; i += 3 * simd_width) {
        simd_acc0 += *reinterpret_cast<const SimdReal*>(&v[i + 0 * simd_width]);
        simd_acc1 += *reinterpret_cast<const SimdReal*>(&v[i + 1 * simd_width]);
        simd_acc2 += *reinterpret_cast<const SimdReal*>(&v[i + 2 * simd_width]);
    }
    for (; i < n; ++i) s += v[i];
    simd_acc0 += simd_acc1 + simd_acc2;
    for (size_t j = 0; j < simd_width; ++j)
        s += reinterpret_cast<const Real*>(&simd_acc0)[j];
    return s;
}
```

Alignment

<table>
<thead>
<tr>
<th>Alignment Type</th>
<th>Address Aligned</th>
<th>Address Unaligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicitly Aligned Load/Store</td>
<td>Fast</td>
<td>Exception Thrown</td>
</tr>
<tr>
<td>Compiled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intrinsic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assembly</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Explicitly Unaligned Load/Store</td>
<td>Fast</td>
<td>Penalty</td>
</tr>
<tr>
<td>Compiled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intrinsic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assembly</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOP + Mem Operand</td>
<td>Fast</td>
<td>Penalty</td>
</tr>
<tr>
<td>Implicitly Unaligned Load</td>
<td>Fast</td>
<td>Penalty</td>
</tr>
</tbody>
</table>
Real fma_simd_unroll3_sum(const Vector &v, size_t n)
{
    const size_t simd_width = sizeof(SimdReal) / sizeof(Real);
    n = std::min(n, v.size());
    Real one = runtime_constant(1.0);
    SimdReal scale;
    for (size_t j = 0; j < simd_width; ++j)
        reinterpret_cast<Real*>(&scale)[j] = one;
    Real s = 0.0;
    size_t i = 0;
    for (; i < n && ((reinterpret_cast<intptr_t>(&v[i]) % 64) != 0); ++i) s += v[i];
    SimdReal simd_acc0 = { 0.0 }; SimdReal simd_acc1 = { 0.0 }; SimdReal simd_acc2 = { 0.0 };
    for (; i + 3 * simd_width <= n; i += 3 * simd_width)
    {
        simd_acc0 += *reinterpret_cast<const SimdReal*>(&v[i + 0 * simd_width]) * scale;
        simd_acc1 += *reinterpret_cast<const SimdReal*>(&v[i + 1 * simd_width]) * scale;
        simd_acc2 += *reinterpret_cast<const SimdReal*>(&v[i + 2 * simd_width]) * scale;
    }
    for ( ; i < n; ++i) s += v[i];
    simd_acc0 += simd_acc1 + simd_acc2;
    for (size_t j = 0; j < simd_width; ++j)
        s += reinterpret_cast<const Real*>(&simd_acc0)[j];
    return s;
}

Real fma_simd_unroll10_sum(const Vector &v, size_t n)
{
    const size_t simd_width = sizeof(SimdReal) / sizeof(Real);
    n = std::min(n, v.size());
    Real one = runtime_constant(1.0);
    SimdReal scale;
    for (size_t j = 0; j < simd_width; ++j)
        reinterpret_cast<Real*>(&scale)[j] = one;
    Real s = 0.0;
    size_t i = 0;
    for (; i < n && ((reinterpret_cast<intptr_t>(&v[i]) % 64) != 0); ++i) s += v[i];
    SimdReal simd_acc0 = { 0.0 };
    // ... 
    SimdReal simd_acc9 = { 0.0 };
    for (; i + 10 * simd_width <= n; i += 10 * simd_width)
    {
        simd_acc9 += *reinterpret_cast<const SimdReal*>(&v[i + 0 * simd_width]) * scale;
        // ...
        simd_acc9 += *reinterpret_cast<const SimdReal*>(&v[i + 9 * simd_width]) * scale;
    }
    for ( ; i < n; ++i) s += v[i];
    simd_acc0 += (simd_acc1 + (simd_acc2 + simd_acc3)) + (simd_acc4 + simd_acc5) + ((simd_acc6 + simd_acc7) + (simd_acc8 + simd_acc9));
    for (size_t j = 0; j < simd_width; ++j)
        s += reinterpret_cast<const Real*>(&simd_acc0)[j];
    return s;
}
Approaches vs Round-Off Errors

naive : n=20000000 sum=-2.613178882234012690e+03 time= 20.878 ms
naive bw : n=20000000 sum=-2.613178882233584318e+03 time= 21.231 ms
naive ur2: n=20000000 sum=-2.613178882233650256e+03 time= 20.558 ms
thread : n=20000000 sum=-2.613178882233670720e+03 time= 20.558 ms
unroll 2 : n=20000000 sum=-2.613178882233953573e+03 time= 15.151 ms
unroll 3 : n=20000000 sum=-2.613178882233649801e+03 time= 15.477 ms
unroll 4 : n=20000000 sum=-2.613178882233655580e+03 time= 15.180 ms
dyn unr 3: n=20000000 sum=-2.613178882233650256e+03 time= 15.065 ms
fma unr 2: n=20000000 sum=-2.613178882233653573e+03 time= 18.226 ms
fma unr 5: n=20000000 sum=-2.613178882233649801e+03 time= 15.477 ms
fma unr10: n=20000000 sum=-2.613178882233655580e+03 time= 15.396 ms
simd : n=20000000 sum=-2.613178882233649801e+03 time= 13.044 ms
misalign : n=20000000 sum=-2.613178882233655580e+03 time= 12.885 ms
simd unr2: n=20000000 sum=-2.613178882233521106e+03 time= 13.360 ms
simd unr3: n=20000000 sum=-2.613178882233670720e+03 time= 13.320 ms
fmasimdui3: n=20000000 sum=-2.613178882233584318e+03 time= 13.247 ms
fmasmdui10: n=20000000 sum=-2.613178882233584318e+03 time= 13.247 ms

Part VI: Optimizing A Classical Coulomb Solver
Coulomb Solver

$1/r$ Long-Range Interactions in $O(N^2)$

Pairwise Interactions
**Coulomb Forces**

**Input**
- charge $q$ of a particle $i$
- position $x_i$ in $\mathbb{R}^3$ with the components $x_i$, $y_i$, and $z_i$

$$x_i = \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}$$

**Distance between two particles**

$$|x_i - x_j| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} \quad (1)$$

**Output**

$$\mathbf{F}_i = q_i \sum_{j=1}^{N} q_j \frac{x_i - x_j}{|x_i - x_j|^3} \quad (i \neq j) \quad \mathbf{F}_i = \begin{pmatrix} F_{x_i} \\ F_{y_i} \\ F_{z_i} \end{pmatrix} \quad (2)$$
FP Instruction Dependencies
Reconstructed from disassembled kernel

- Addition, Multiplication: fully pipelined, fixed latency
- Division, Square root: not pipelined, latency depends on the data type (7, 14, 28 cycles)

The critical path

- 10 instructions: longest sequence of dependent instructions
- 90 cycles: lower bound on the latency of the inner loop
- $56 = 2 \cdot 28$ cycles blocking the divider unit