JÜLICH
FORSCHUNGSZENTRUM

Vectorization

For Non-Trivial Datastructures in C++

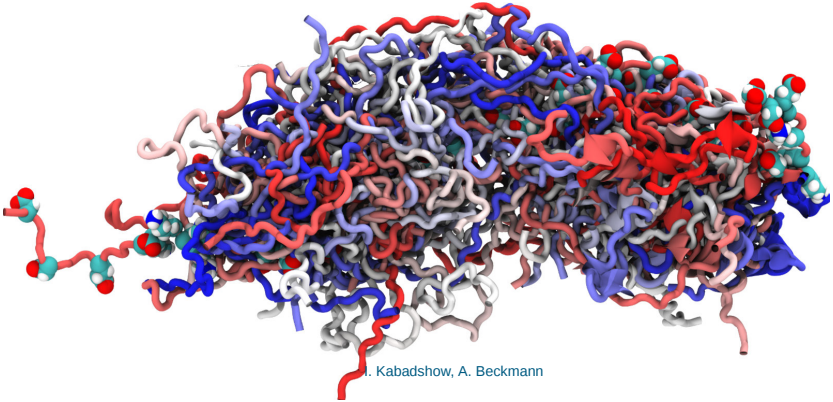
September 28th, 2017 | Ivo Kabadshow & Andreas Beckmann | IXPUG 2017, Austin, Texas, USA

Motivation

Coulomb problem: $1/r$ potential and the extension to λ-dynamics for GROMACS

Task: Compute all pairwise interactions of N particles

N-body problem: $\mathcal{O}(N^2) \rightarrow \mathcal{O}(N)$ with FMM



Motivation

Coulomb problem: $1/r$ potential and the extension to λ-dynamics for GROMACS

Task: Compute all pairwise interactions of N particles

N-body problem: $\mathcal{O}(N^2) \rightarrow \mathcal{O}(N)$ with FMM

Why is that an issue?

- MD targets $<$ 1ms runtime per time step
- support for many platforms needed (SSE, AVX2, AVX512, ARM, Power)
- not compute-bound, but synchronization bound
- no libraries (like BLAS) to do the heavy lifting

We might have to look under the hood ... and get our hands dirty.

Relevance for this audience

Many more cores	4×

- Loop-level parallelism vs. tasking
- Critical path analysis
- Cache coherence: scalable locks?
- NUMA effects: static vs. dynamic load balancing

Relevance for this audience

Many more cores — 4×

- Loop-level parallelism vs. tasking
- Critical path analysis
- Cache coherence: scalable locks?
- NUMA effects: static vs. dynamic load balancing

Wider SIMD vector — 2×

- Data layout and data access pattern
- Portability?

Portable Vectorization with C++11?

Readability, Maintainability

One Kernel To Rule Them All?

- Write the compute kernel once (high-level C++11)
- Reuse it with different precisions (float, double)
- Reuse it on different SIMD widths

Portable Vectorization with C++11?

Readability, Maintainability

One Kernel To Rule Them All?

- Write the compute kernel once (high-level C++11)
- Reuse it with different precisions (float, double)
- Reuse it on different SIMD widths

Non-trivial questions

- Does the compiler already vectorize for me?
- How does it look on a different platform?
- Can we introduce a portable abstraction?
- Does that introduce overhead?

Portable Vectorization with C++11?

Readability, Maintainability

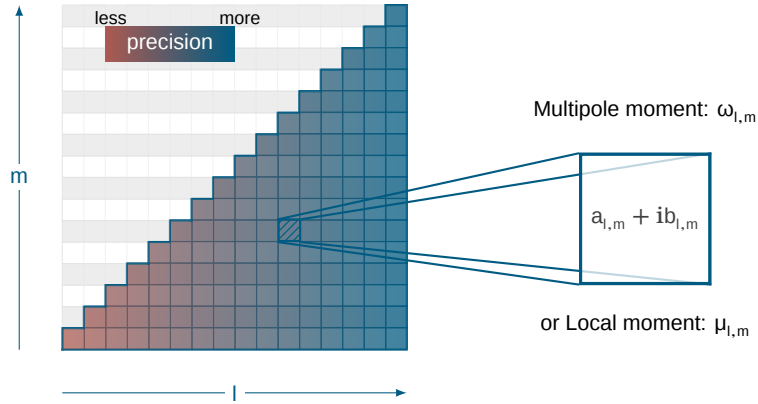
One Kernel To Rule Them All?

- Write the compute kernel once (high-level C++11)
- Reuse it with different precisions (float, double)
- Reuse it on different SIMD widths

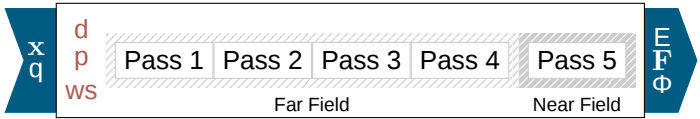
What should I do, if the datastructure is not trivial?

Non-Trivial Datastructures

Triangular Array hold Multipole or Local Moments: $0 \leq p \leq 50, 0 \leq l \leq p, 0 \leq m \leq l$

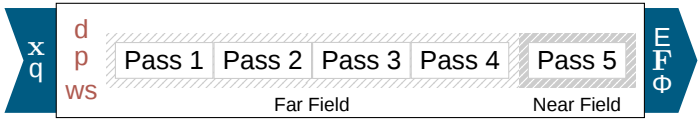


FMM Workflow



FMM Pass	SIMD ease
Pass 1: operations between particles and triangular arrays	★★★
Pass 1: operations between triangular arrays	★☆☆
Pass 2: operations between triangular arrays	★☆☆
Pass 3: operations between triangular arrays	★☆☆
Pass 4: operations between triangular arrays and particles	★★★
Pass 5: operations between particles	★★☆

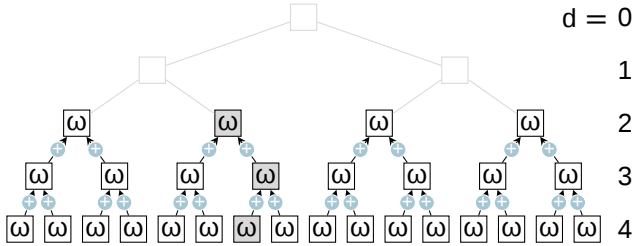
FMM Workflow



FMM Pass	SIMD ease
Pass 1: operations between particles and triangular arrays	★★★
Pass 1: operations between triangular arrays	★☆☆
Pass 2: operations between triangular arrays	★☆☆
Pass 3: operations between triangular arrays	★☆☆
Pass 4: operations between triangular arrays and particles	★★★
Pass 5: operations between particles	★★☆

FMM Workflow

Pass 1: multipole to multipole, shifting multipoles upwards

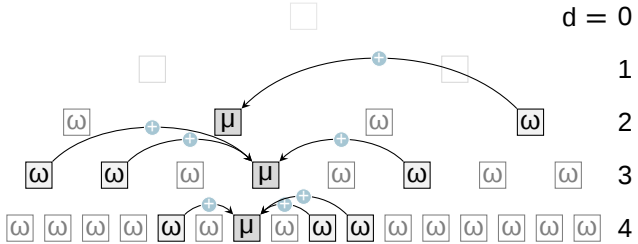


Parallelism

- 8^d operations per depth

FMM Workflow

Pass 2: multipole to local, translate remote multipoles into local Taylor moments

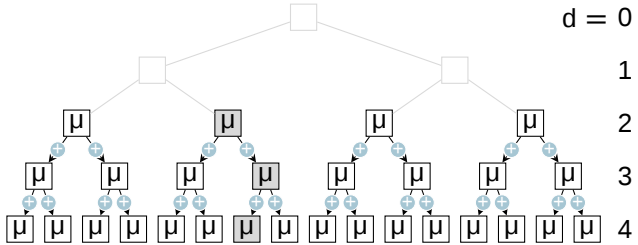


Parallelism

- 189×8^d operations per depth

FMM Workflow

Pass 3: local to local, shifting Taylor moments downwards



Parallelism

- 8^d operations per depth

Reuse between the FMM Passes

M2M, M2L and L2L have the same algorithmic workflow

Operations from the different passes share code paths

1. Forward-rotation of input $\mathcal{O}(p^3)$
2. Shift of rotated input to rotated output $\mathcal{O}(p^3)$
3. Backward-rotation of rotated output back to output $\mathcal{O}(p^3)$

- Memory layout: column-major/row-major
- Temporaries: choose optimal layout for input of the subsequent operation

Forward-Rotation Operator

Source Code

Rotation operator loop structure — $\mathcal{O}(p^3)$ version

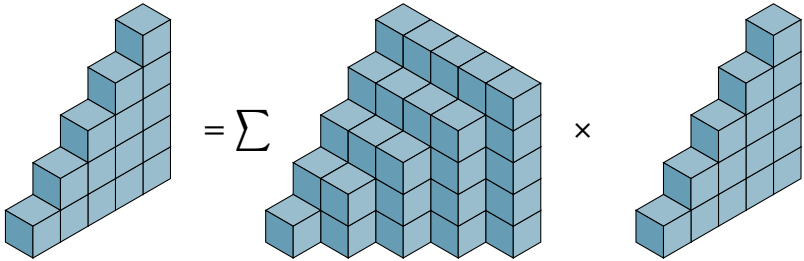
```
for (l = 0; l <= p; ++l)
  for (m = 0; m <= l; ++m)
    for (k = 0; k <= l; ++k)
      input_rot[l, m] += rot[l, m, k] * input[l, -k]
```

- $\mathcal{O}(p^2)$ terms have been omitted
- Datatypes: real (float, double, ...), complex<real>

⚠ `rot` scales real and imag parts differently

Classical Forward-Rotation Operation

No reuse of rotation matrix (middle)



Forward-Rotation Operator (innermost loop)

Compiler-Generated Code

```
loop:
vmovss (%rax,%r10,8),%xmm6
vmovss 0x4(%rax,%r10,8),%xmm7
vfmadd231ss (%rsi,%r10,8),%xmm6,%xmm0
vfmadd231ss 0x4(%rsi,%r10,8),%xmm7,%xmm1
lea    0x1(%r10),%r10
cmp    %rdx,%r10
jle    loop
```

Forward-Rotation Operator (innermost loop)

Compiler-Generated Code

```
loop:
vmovss (%rax,%r10,8),%xmm6
vmovss 0x4(%rax,%r10,8),%xmm7
vfmadd231ss (%rsi,%r10,8),%xmm6,%xmm0
vfmadd231ss 0x4(%rsi,%r10,8),%xmm7,%xmm1
lea    0x1(%r10),%r10
cmp    %rdx,%r10
jle    loop
```

Naive Approach

Vectorizing the Innermost Loop

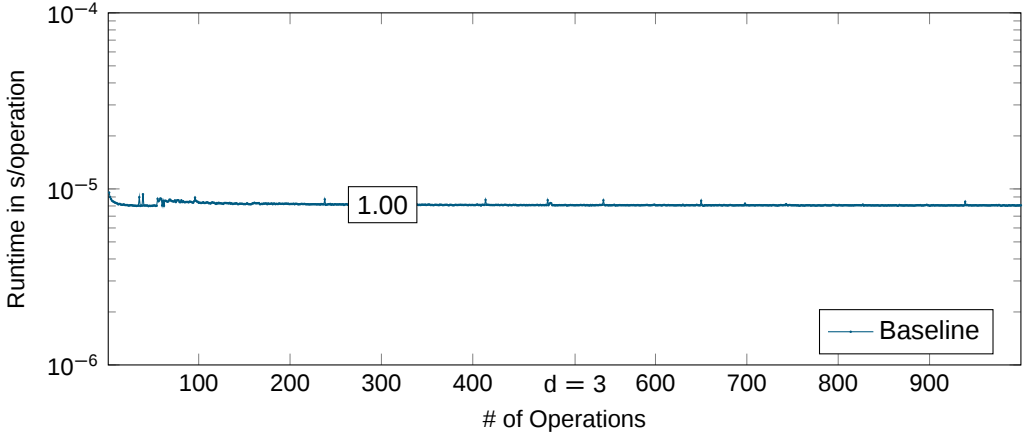
- Iteration counts: 1, … , p
- Needs padding to SIMD width or special handling of remaining iterations

Unrolling the Middle Loop

- Improves reuse of single input (triangular array)

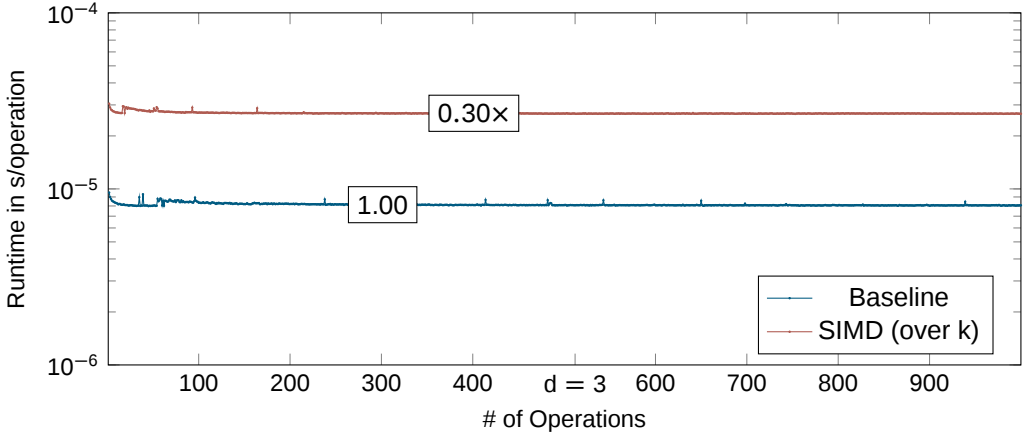
Baseline Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



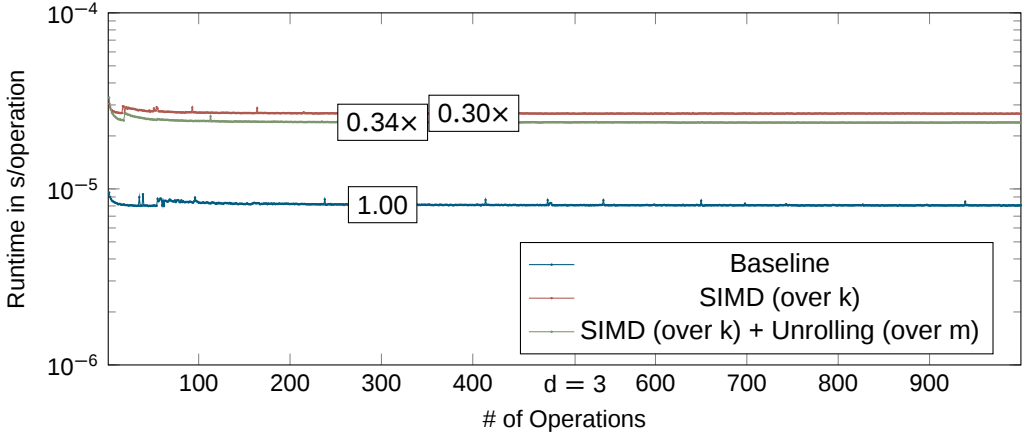
Baseline Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



Baseline Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



What is the bottleneck?

Instruction latency

- 5 cycles latency for FMA
- Microarchitecture can issue 2 FMA per cycle

Memory bandwidth

- Only a single FLOP (FMA) per (two) memory load(s)
- Rotation matrix is large $\mathcal{O}(p^3)$
- No reuse of rotation matrix for single input $\mathcal{O}(p^2)$ possible

Better Approach

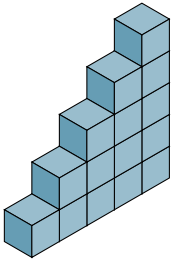
Unrolling over multiple inputs (triangular arrays)

- Reuse of the rotation matrix
- Stack a fixed amount of triangular arrays elementwise: AoS to SoA
- Create a triangular array of complex numbers of stacks
- Temporary reordering/construction of input and output $\mathcal{O}(p^2)$
- Can be done on-the-fly: Permanent storage in stacked memory layout not useful
- Only a single fixed permutation storable, but many needed for computation
- Arithmetic operations on stacks will be element-wise

Datalayout

What we have

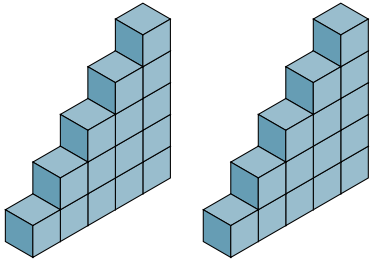
- Struct of Complex (SoC) → `RI RI RI`



Datalayout

What we have

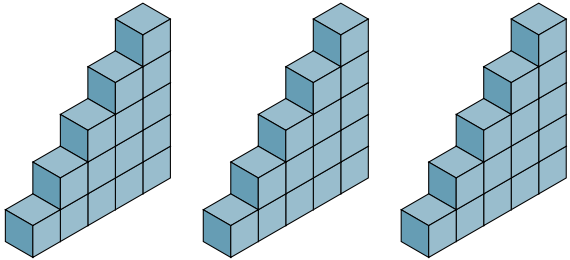
- Array of Struct of Complex (AoSoC) …



Datalayout

What we have

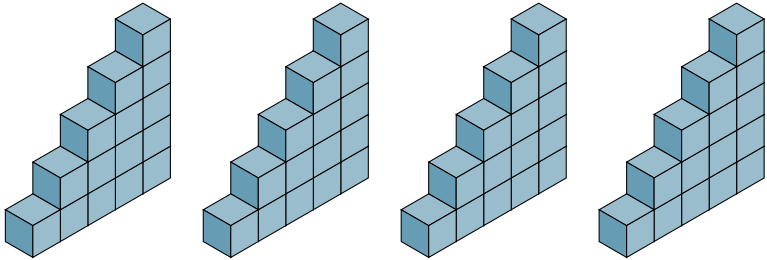
- Array of Struct of Complex (AoSoC) …



Datalayout

What we have

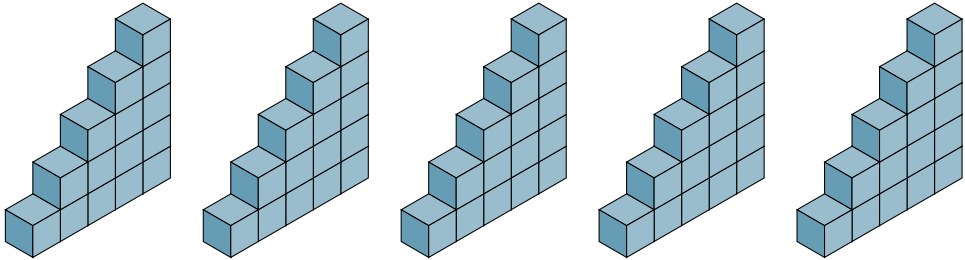
- Array of Struct of Complex (AoSoC) …



Datalayout

What we have

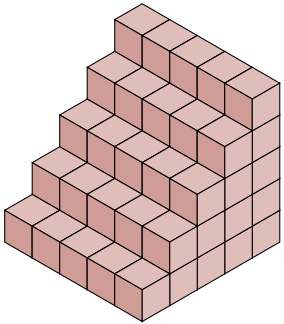
- Array of Struct of Complex (AoSoC) …



Datalayout

What we need

- Struct of Complex of Array (SoCoA) → `RRRRRIIIII`



Stacked Forward-Rotation Operator

Rotation operator loop structure — $\mathcal{O}(p^3)$ version

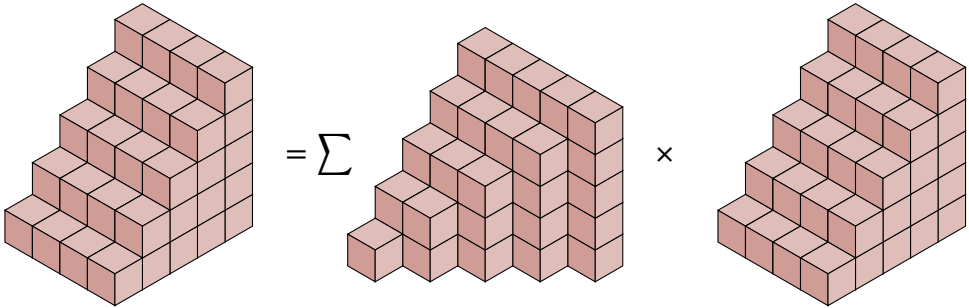
```
for (l = 0; l <= p; ++l)
  for (m = 0; m <= l; ++m)
    for (k = 0; k <= l; ++k)
      input_rot[l, m] += rot[l, m, k] * input[l, -k]
```

- $\mathcal{O}(p^2)$ terms have been omitted
- Datatypes: real (float, double, ...), stack<real>, complex<stack<real> >

⚠ `rot` scales real and imag parts differently

Stacked Forward-Rotation Operation

Stack=4 via template parameter



Stacked Forward-Rotation Operation

Compiler-Generated Code, Stack=4

```
loop:
vmovss 0x4(%rdi,%rcx,8),%xmm8
lea    0x20(%rax),%rax
vmovss (%rdi,%rcx,8),%xmm9
lea    0x1(%rcx),%rcx
cmp    %rdx,%rcx
vfmadd231ss -0x20(%rax),%xmm9,%xmm0
vfmadd231ss -0x1c(%rax),%xmm9,%xmm5
vfmadd231ss -0x18(%rax),%xmm9,%xmm7
vfmadd231ss -0x14(%rax),%xmm9,%xmm2
vfmadd231ss -0x10(%rax),%xmm8,%xmm4
vfmadd231ss -0xc(%rax),%xmm8,%xmm1
vfmadd231ss -0x8(%rax),%xmm8,%xmm6
vfmadd231ss -0x4(%rax),%xmm8,%xmm3
jle    loop
```

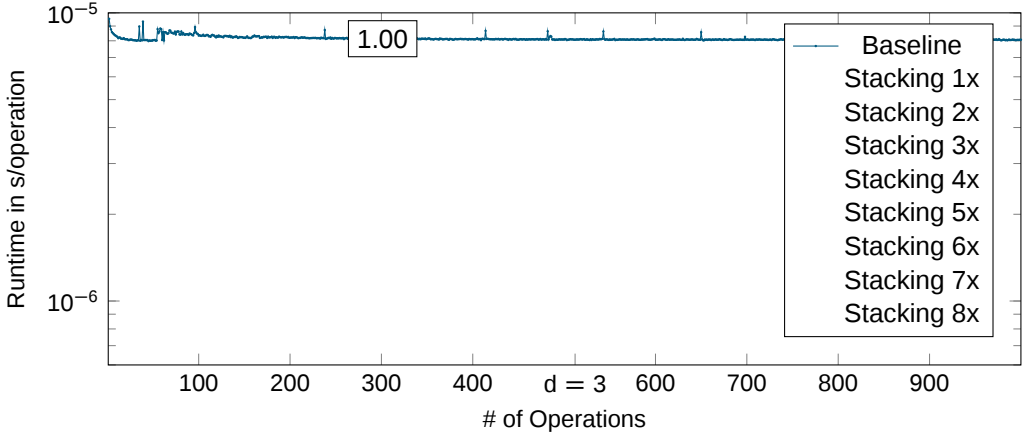
Stacked Forward-Rotation Operation

Compiler-Generated Code, Stack=4

```
loop:
vmovss 0x4(%rdi,%rcx,8),%xmm8
lea    0x20(%rax),%rax
vmovss (%rdi,%rcx,8),%xmm9
lea    0x1(%rcx),%rcx
cmp    %rdx,%rcx
vfmadd231ss -0x20(%rax),%xmm9,%xmm0
vfmadd231ss -0x1c(%rax),%xmm9,%xmm5
vfmadd231ss -0x18(%rax),%xmm9,%xmm7
vfmadd231ss -0x14(%rax),%xmm9,%xmm2
vfmadd231ss -0x10(%rax),%xmm8,%xmm4
vfmadd231ss -0xc(%rax),%xmm8,%xmm1
vfmadd231ss -0x8(%rax),%xmm8,%xmm6
vfmadd231ss -0x4(%rax),%xmm8,%xmm3
jle    loop
```

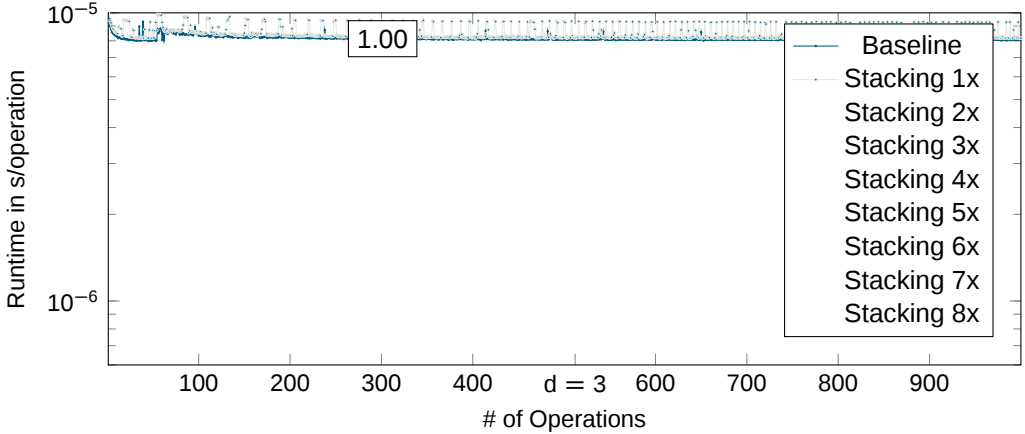
Basic Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



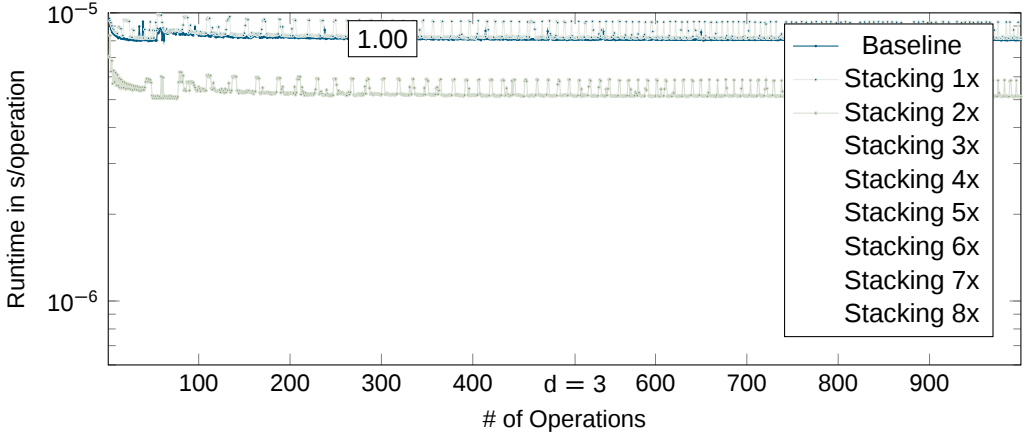
Basic Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



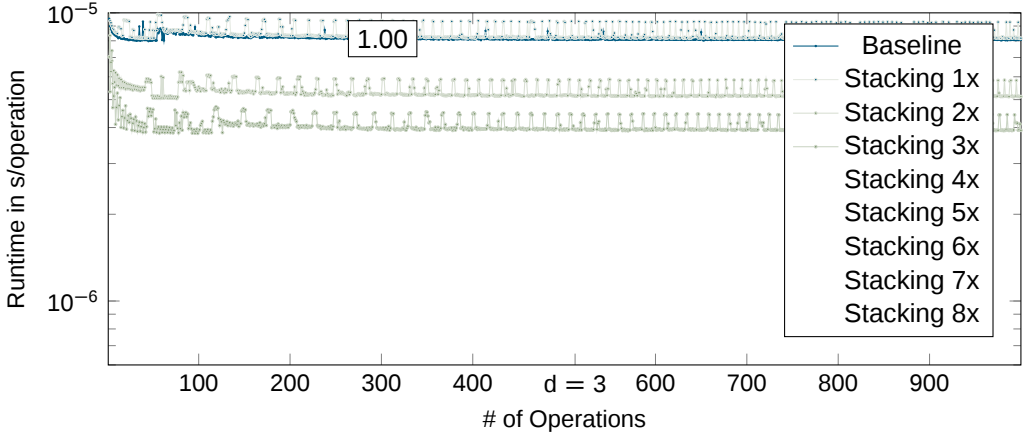
Basic Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



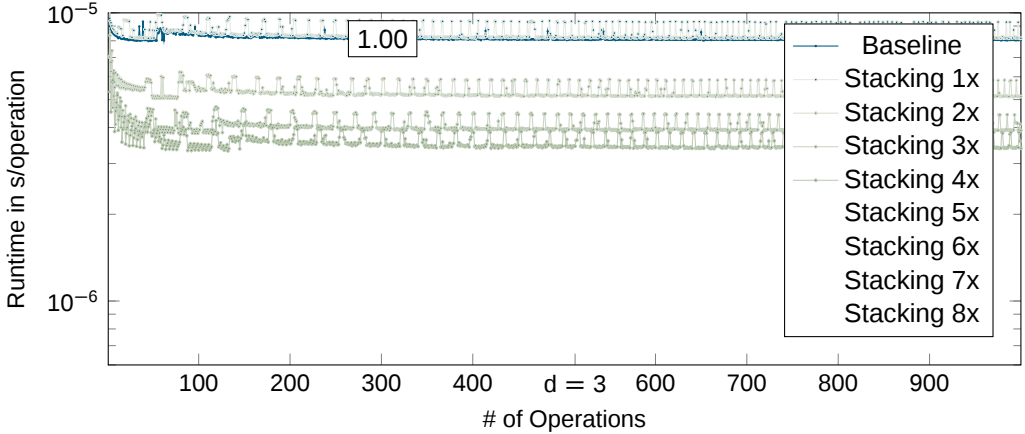
Basic Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



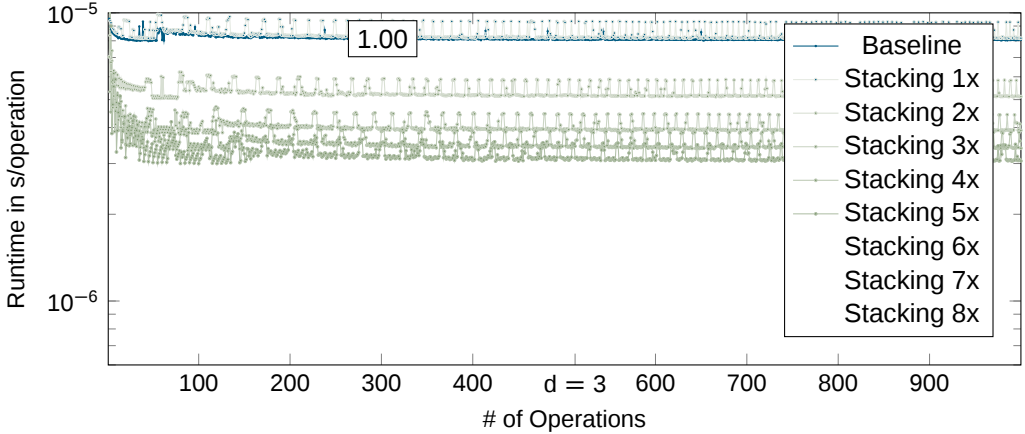
Basic Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



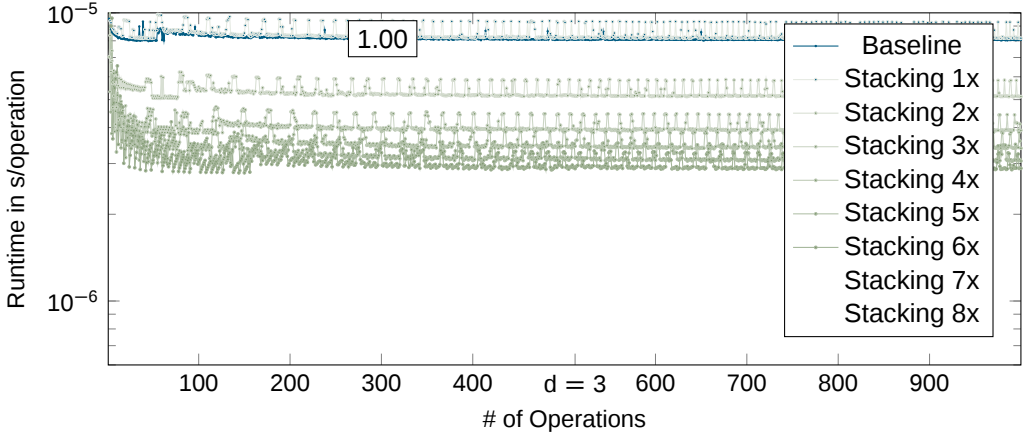
Basic Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



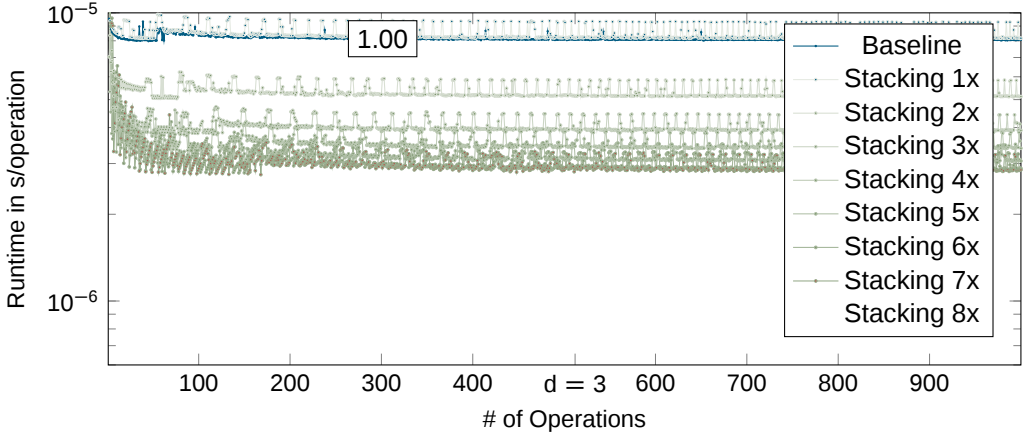
Basic Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



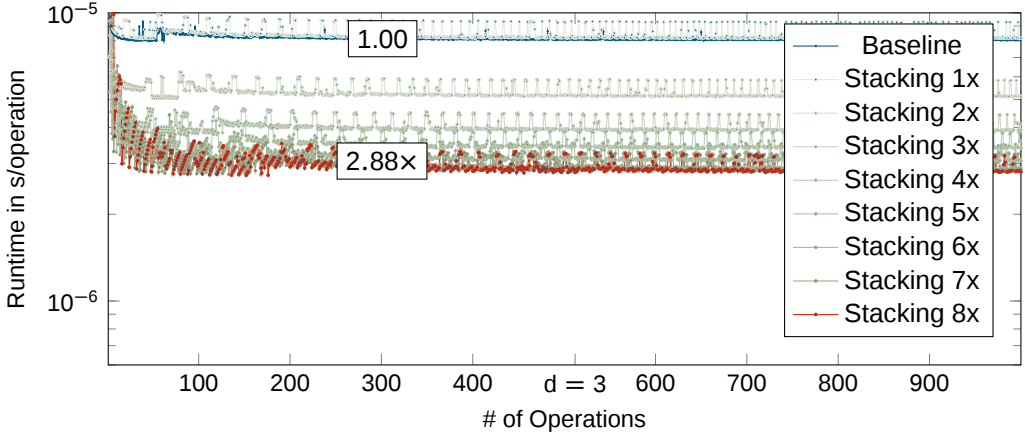
Basic Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float

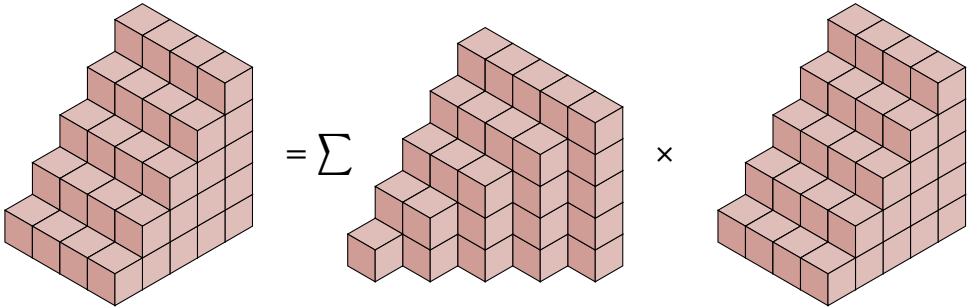


Basic Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



Stacked SIMD Forward-Rotation Operation



Stacked SIMD Forward-Rotation Operation

Compiler-Generated Code, SIMD=16, Stack=1

```
loop:
vbroadcastss (%rcx,%rax,8),%zmm4
lea    0x80(%rdx),%rdx
vfmadd231ps -0x80(%rdx),%zmm4,%zmm2
vbroadcastss 0x4(%rcx,%rax,8),%zmm4
lea    0x1(%rax),%rax
vfmadd231ps -0x40(%rdx),%zmm4,%zmm3
cmp    %rdi,%rax
jle    loop
```

Stacked SIMD Forward-Rotation Operation

Compiler-Generated Code, SIMD=16, Stack=1

```
loop:
vbroadcastss (%rcx,%rax,8),%zmm4
lea    0x80(%rdx),%rdx
vfmadd231ps -0x80(%rdx),%zmm4,%zmm2
vbroadcastss 0x4(%rcx,%rax,8),%zmm4
lea    0x1(%rax),%rax
vfmadd231ps -0x40(%rdx),%zmm4,%zmm3
cmp    %rdi,%rax
jle    loop
```

Stacked SIMD Forward-Rotation Operation

Compiler-Generated Code, SIMD=16, Stack=2

```
loop:
vbroadcastss 0x4(%rcx,%rdx,8),%zmm0
lea    0x100(%rax),%rax
vbroadcastss (%rcx,%rdx,8),%zmm1
lea    0x1(%rdx),%rdx
cmp    %r10,%rdx
vfmadd231ps -0x100(%rax),%zmm1,%zmm5
vfmadd231ps -0xc0(%rax),%zmm1,%zmm2
vfmadd231ps -0x80(%rax),%zmm0,%zmm4
vfmadd231ps -0x40(%rax),%zmm0,%zmm3
jle    loop
```

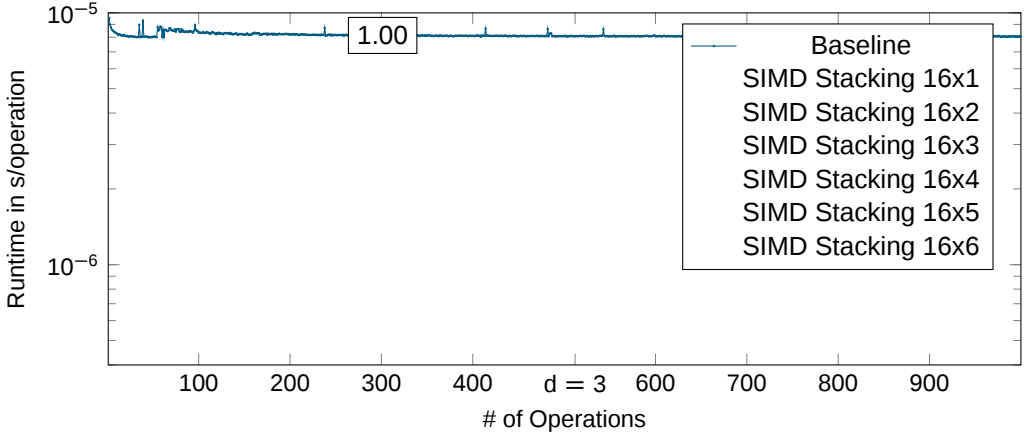
Stacked SIMD Forward-Rotation Operation

Compiler-Generated Code, SIMD=16, Stack=2

```
loop:
vbroadcastss 0x4(%rcx,%rdx,8),%zmm0
lea    0x100(%rax),%rax
vbroadcastss (%rcx,%rdx,8),%zmm1
lea    0x1(%rdx),%rdx
cmp    %r10,%rdx
vfmadd231ps -0x100(%rax),%zmm1,%zmm5
vfmadd231ps -0xc0(%rax),%zmm1,%zmm2
vfmadd231ps -0x80(%rax),%zmm0,%zmm4
vfmadd231ps -0x40(%rax),%zmm0,%zmm3
jle    loop
```

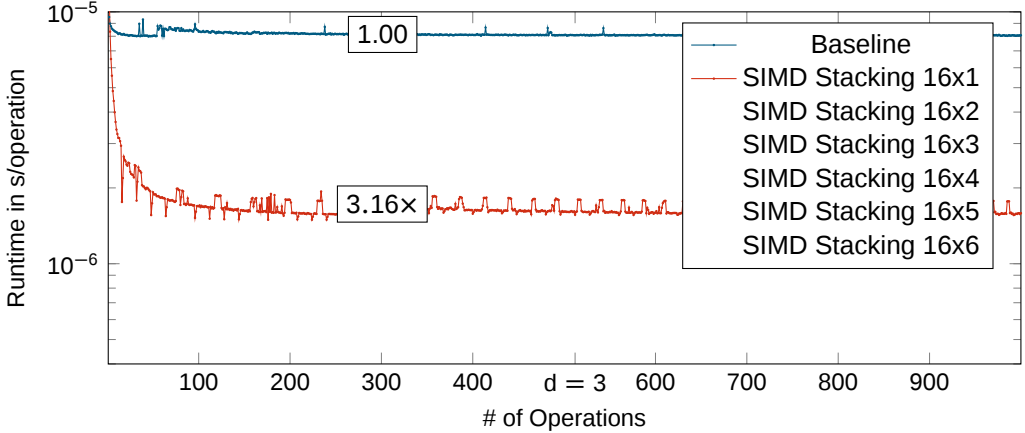
SIMD Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



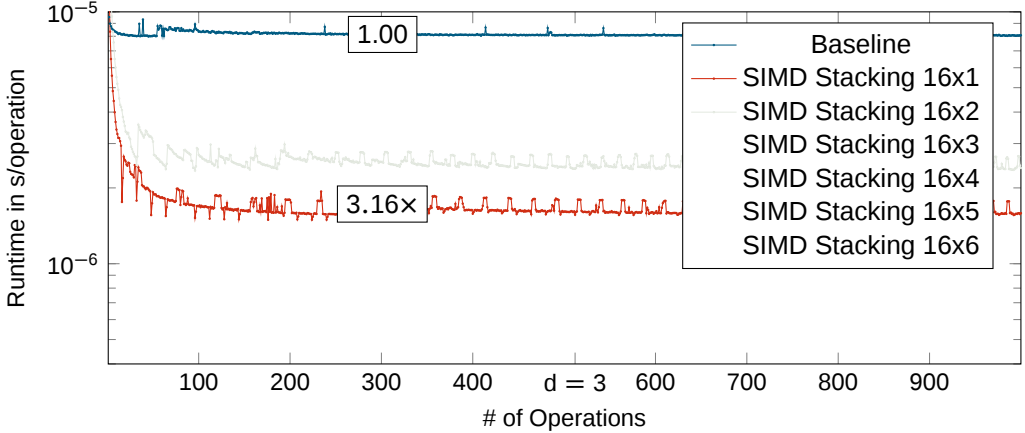
SIMD Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



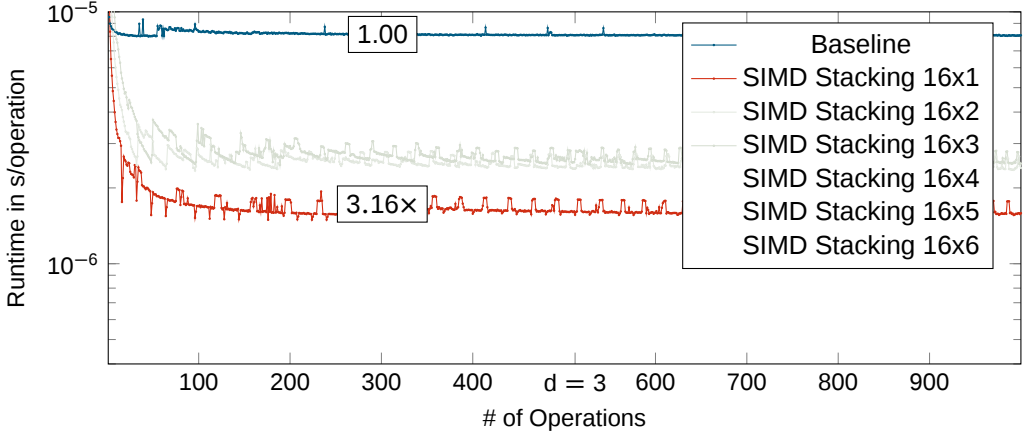
SIMD Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



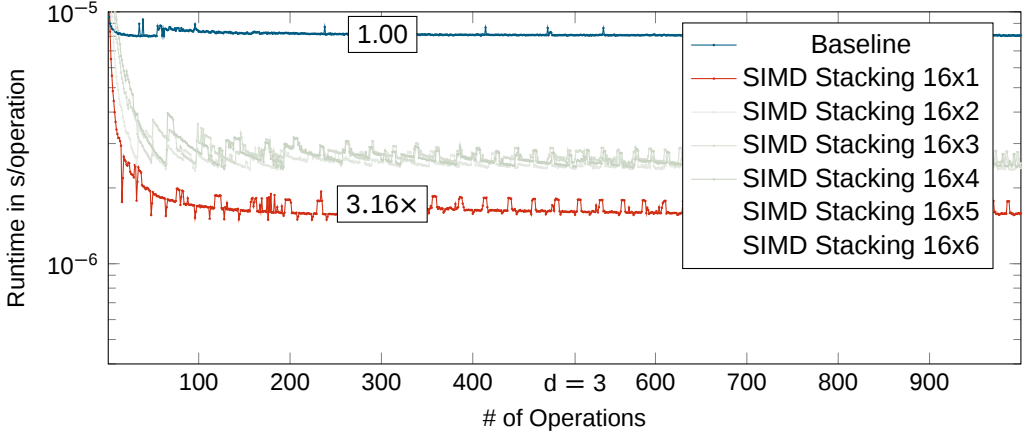
SIMD Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



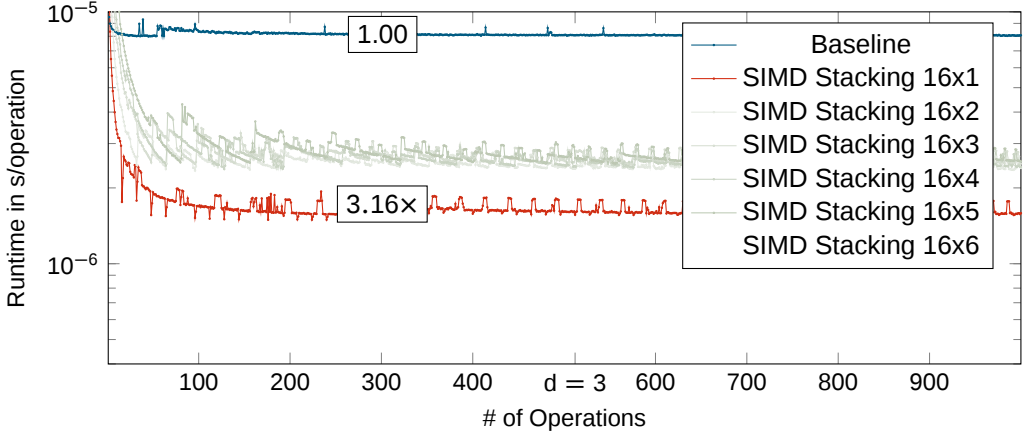
SIMD Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



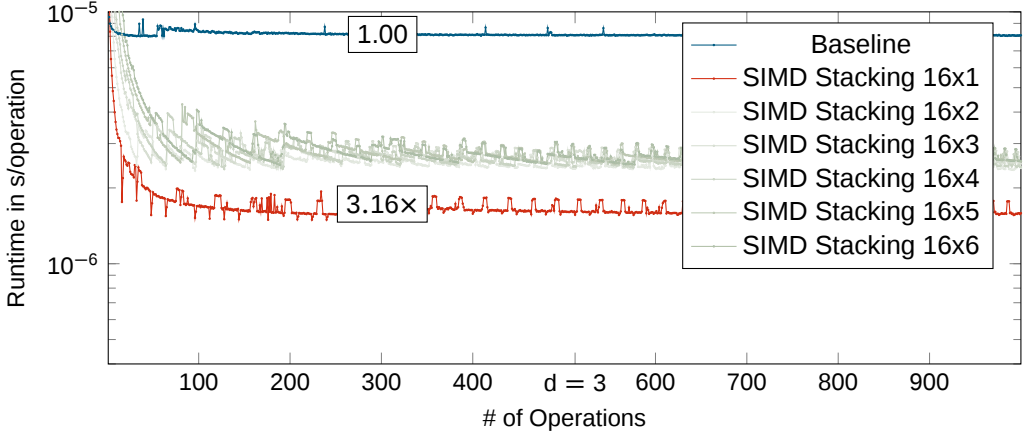
SIMD Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



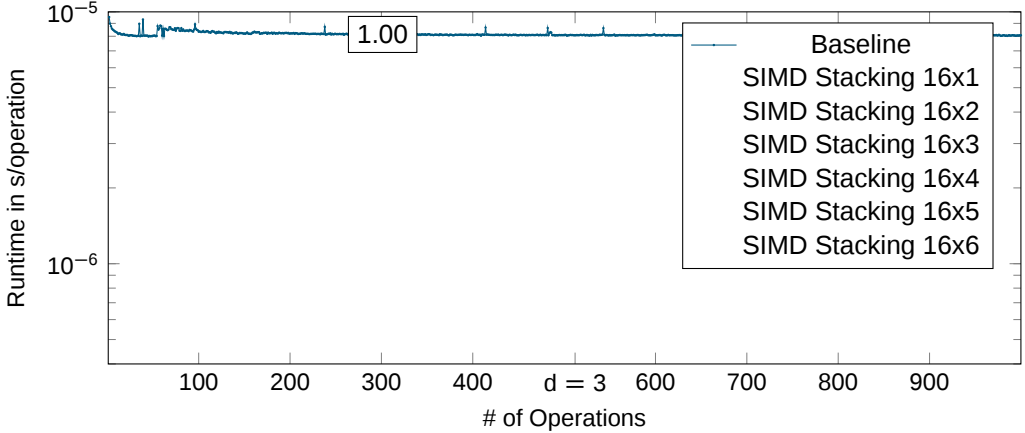
SIMD Stacking Timings p = 10

Jurecabooster: Xeon-Phi-7250F, float



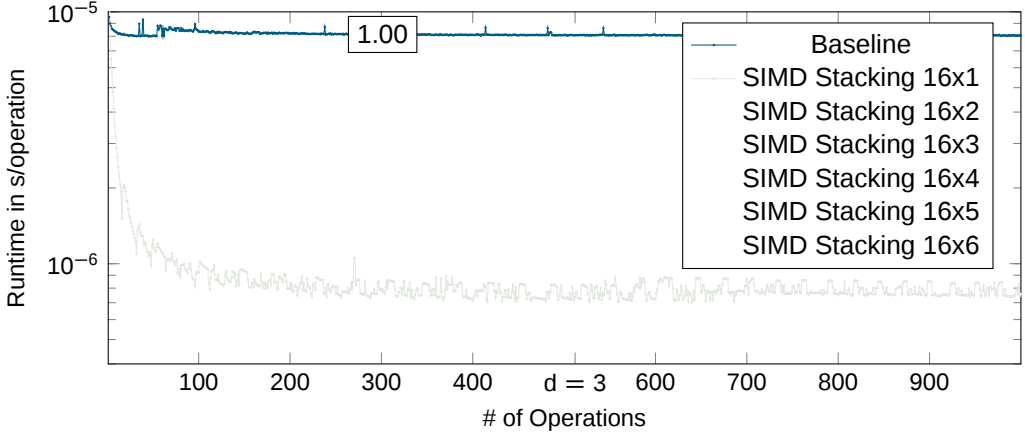
SIMD Stacking Timings $p = 10$ Without Reordering Costs

Jurecabooster: Xeon-Phi-7250F, float



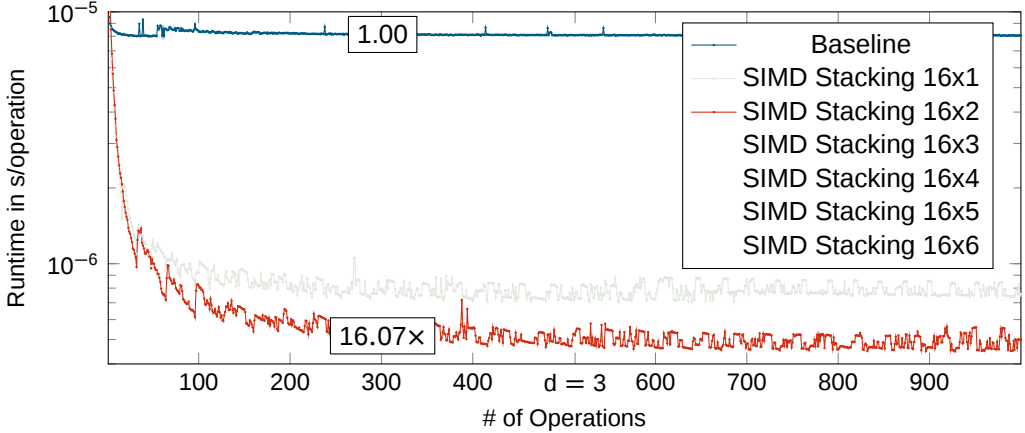
SIMD Stacking Timings $p = 10$ Without Reordering Costs

Jurecabooster: Xeon-Phi-7250F, float



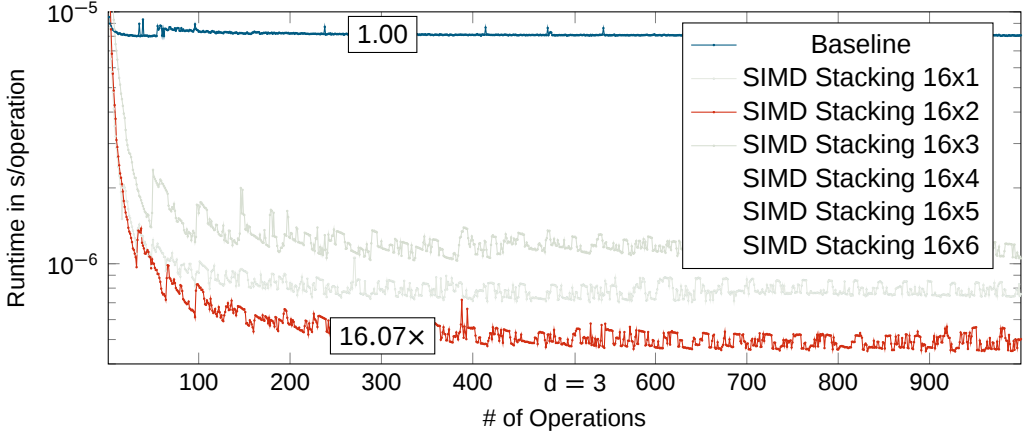
SIMD Stacking Timings p = 10 Without Reordering Costs

Jurecabooster: Xeon-Phi-7250F, float



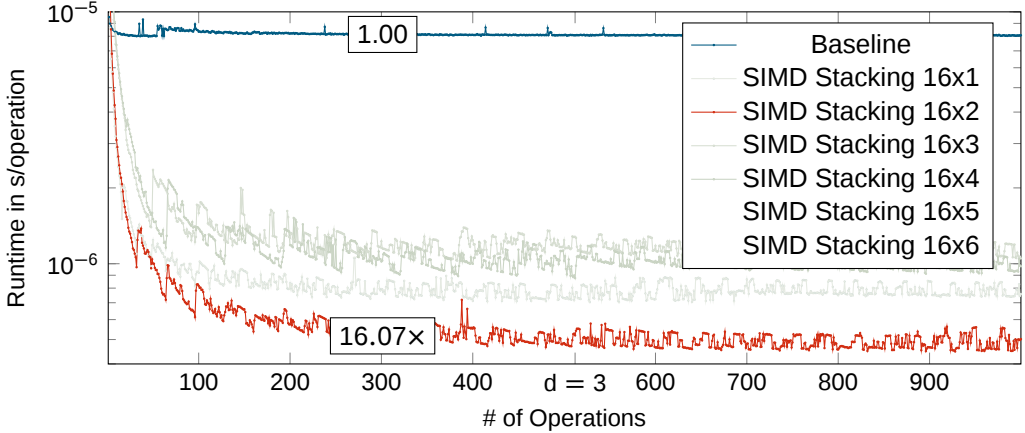
SIMD Stacking Timings p = 10 Without Reordering Costs

Jurecabooster: Xeon-Phi-7250F, float



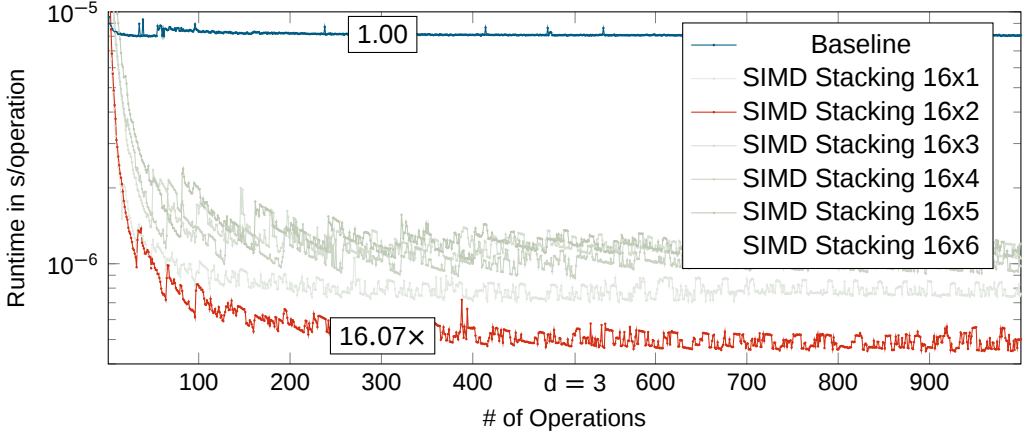
SIMD Stacking Timings $p = 10$ Without Reordering Costs

Jurecabooster: Xeon-Phi-7250F, float



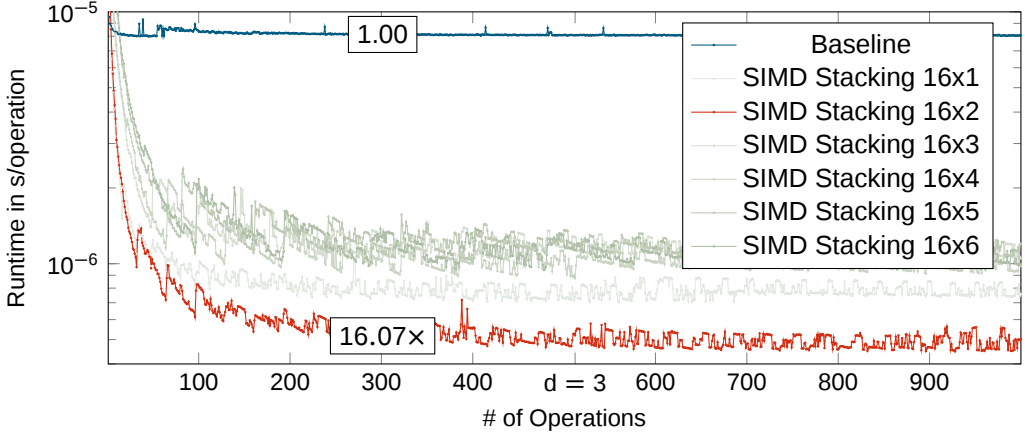
SIMD Stacking Timings p = 10 Without Reordering Costs

Jurecabooster: Xeon-Phi-7250F, float



SIMD Stacking Timings p = 10 Without Reordering Costs

Jurecabooster: Xeon-Phi-7250F, float



What to do for a new SIMD platform?

Change 90 lines of generic code, compiler does the details

High-level C++ code

- Partial load
- Init broadcast `{a,a,a,a}`
- Transpose block

What to do for a new SIMD platform?

Change 90 lines of generic code, compiler does the details

Wrapper over low-level Intrinsics: e.g AVX-512

- Unaligned load: `_mm512_loadu_ps(p)`
- Square root `_mm512_sqrt_ps(a)`

What to do on a new SIMD platform II

Benchmark Parameter Space

- Check SIMD for different precisions (float, double)
- Check for additional stacking (reuse)
- Check for additional unrolling (reuse)
- Use different compiler

Find optimal parameter set

- Store optimal settings as platform configuration
- May depend on expansion length p

Portable Compute Kernel

Sequential, Stacked, SIMD, SIMD+Stacked Code

```
template <typename TA_in, typename Rot, typename TA_out>
void RotationLocalBackward(
    const TA_in &mu_in_rot, const Rot &R, TA_out &mu_out, const size_t p)
{
    for (size_t l = 0; l <= p; ++l) {
        for (size_t m = 0; m <= l; ++m) {
            typename TA_out::value_type mu = 0.0;
            for (size_t k = 1; k <= l; ++k)
                mu += scale_complex(R.d_g(l, m, k), mu_in_rot(l, k));
            mu *= R.e_imphi[m];
            mu_out(l, m) += mu;
        }
    }
}
```

Conclusion

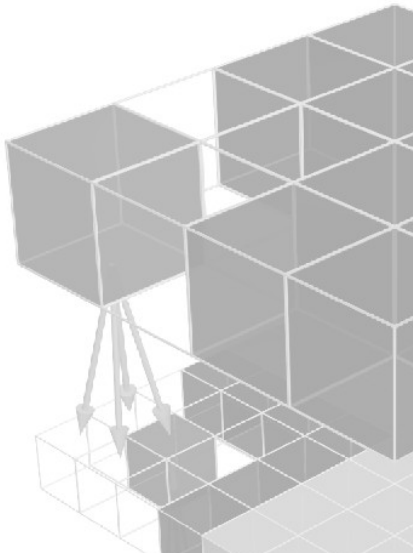
- One code for scalar and arbitrarily stacked/vectorized usage
- Input triangular stacking helps reuse of rotation matrix
- Stacking generates more ILP and reduces FMA completion stalls
- Abstraction can handle arbitrary SIMD widths (even prime)

- Stacking/SIMD need overloads of the arithmetic operations
- On-the-fly p^2 reordering reduces effectiveness of p^3 compute for small p
- Very large SIMD stacks need more care, compiler needs a how-to advise

Thank you for your attention.

Questions?

Contact: Ivo Kabadshow
i.kabadshow@fz-juelich.de



GCC vs. ICC

Or why we don't use the Intel Compiler

Compiler versions

- g++ (GCC) 7.2.0
- icpc (ICC) 17.0.2 20170213

Common compiler flags

- `-std=c++11 -O3 -march=native -W -Wall -g -MD -fopenmp`

Additional GCC flags

- `-Wno-ignored-attributes -fabi-version=0 -fext-numeric-literals`

ICC Assembly

SIMD=16, Stack=1

```
loop:
mov    %r10,%r14
vmovups (%r11,%r15,1),%zmm5
vmovups 0x40(%r11,%r15,1),%zmm8
add    $0x80,%r11
add    0x0(%r13,%rdx,8),%r14
add    %rax,%r14
add    $0x1,%rax
mov    (%r12,%r14,8),%ebx
vmovd  %ebx,%xmm3
mov    0x4(%r12,%r14,8),%ebx
vbroadcastss %xmm3,%zmm4
vmovd  %ebx,%xmm6
vbroadcastss %xmm6,%zmm7
vmulps %zmm5,%zmm4,%zmm9
vmulps %zmm8,%zmm7,%zmm10
vaddps %zmm9,%zmm1,%zmm1
vmovups %zmm1,0xc0(%rsp)
vaddps %zmm10,%zmm0,%zmm0
vmovups %zmm0,0x100(%rsp)
cmp    %rdx,%rax
jle    loop
```

GCC Assembly

SIMD=16, Stack=1

```
loop:
vbroadcastss (%rcx,%rax,8),%zmm4
lea    0x80(%rdx),%rdx
vfmadd231ps -0x80(%rdx),%zmm4,%zmm2
vbroadcastss 0x4(%rcx,%rax,8),%zmm4
lea    0x1(%rax),%rax
vfmadd231ps -0x40(%rdx),%zmm4,%zmm3
cmp    %rdi,%rax
jle    loop
```

GCC Assembly

SIMD=16, Stack=1

```
loop:
vbroadcastss (%rcx,%rax,8),%zmm4
lea    0x80(%rdx),%rdx
vfmadd231ps -0x80(%rdx),%zmm4,%zmm2
vbroadcastss 0x4(%rcx,%rax,8),%zmm4
lea    0x1(%rax),%rax
vfmadd231ps -0x40(%rdx),%zmm4,%zmm3
cmp    %rdi,%rax
jle    loop
```

ICC Assembly

SIMD=16, Stack=2

```
loop:
mov    %rbx,%r13
vmovups 0x100(%r12,%rcx,1),%zmm2
vmovups 0x140(%r12,%rcx,1),%zmm4
add    (%r15,%rax,8),%r13
vmovups 0x180(%r12,%rcx,1),%zmm6
add    %r11,%r13
vmovups 0x1c0(%r12,%rcx,1),%zmm8
add    $0x1,%r11
add    $0x100,%r12
mov    0x8(%r14,%r13,8),%edx
vmovd  %edx,%xmm1
mov    0xc(%r14,%r13,8),%edx
vbroadcastss %xmm1,%zmm3
vmovd  %edx,%xmm5
vbroadcastss %xmm5,%zmm7
vmulps %zmm2,%zmm3,%zmm9
vmulps %zmm4,%zmm3,%zmm11
vmulps %zmm6,%zmm7,%zmm13
vmulps %zmm8,%zmm7,%zmm15
vaddps 0x400(%rsp),%zmm9,%zmm10
vmovups %zmm10,0x400(%rsp)
vaddps 0x440(%rsp),%zmm11,%zmm12
vmovups %zmm12,0x440(%rsp)
vaddps 0x480(%rsp),%zmm13,%zmm14
vmovups %zmm14,0x480(%rsp)
vaddps 0x4c0(%rsp),%zmm15,%zmm16
vmovups %zmm16,0x4c0(%rsp)
cmp    %rax,%r11
jb     loop
```

GCC Assembly

SIMD=16, Stack=2

```
loop:
vbroadcastss 0x4(%rcx,%rdx,8),%zmm0
lea    0x100(%rax),%rax
vbroadcastss (%rcx,%rdx,8),%zmm1
lea    0x1(%rdx),%rdx
cmp    %r10,%rdx
vfmadd231ps -0x100(%rax),%zmm1,%zmm5
vfmadd231ps -0xc0(%rax),%zmm1,%zmm2
vfmadd231ps -0x80(%rax),%zmm0,%zmm4
vfmadd231ps -0x40(%rax),%zmm0,%zmm3
jle    loop
```

GCC Assembly

SIMD=16, Stack=2

```
loop:
vbroadcastss 0x4(%rcx,%rdx,8),%zmm0
lea    0x100(%rax),%rax
vbroadcastss (%rcx,%rdx,8),%zmm1
lea    0x1(%rdx),%rdx
cmp    %r10,%rdx
vfmadd231ps -0x100(%rax),%zmm1,%zmm5
vfmadd231ps -0xc0(%rax),%zmm1,%zmm2
vfmadd231ps -0x80(%rax),%zmm0,%zmm4
vfmadd231ps -0x40(%rax),%zmm0,%zmm3
jle    loop
```

JÜLICH
FORSCHUNGSZENTRUM

Vectorization

For Non-Trivial Datastructures in C++

September 28th, 2017 | Ivo Kabadshow & Andreas Beckmann | IXPUG 2017, Austin, Texas, USA