SIMD Vectorization in Julia in the Context of Nuclear Reactor Simulation

Speaker: John Tramm

10/24/2016

SIMD: Critical for High Performance

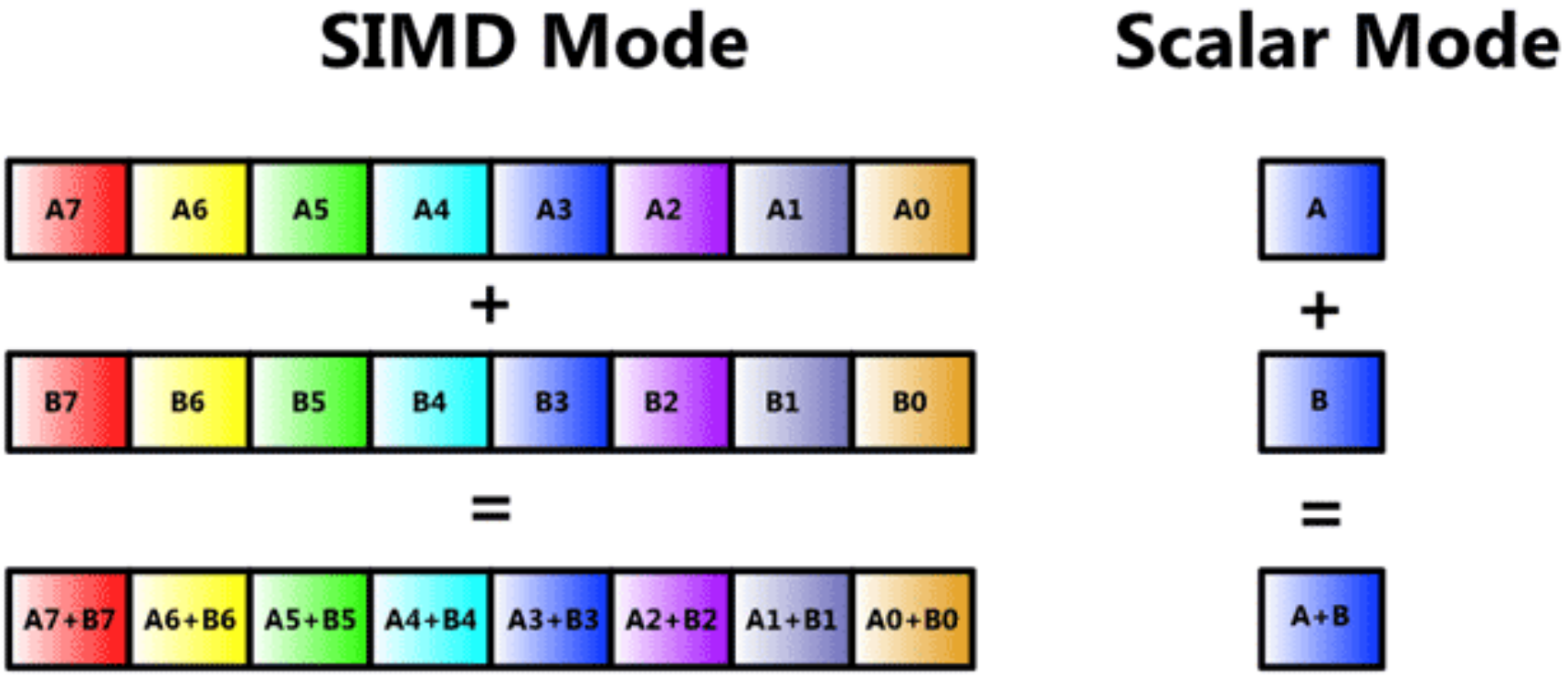
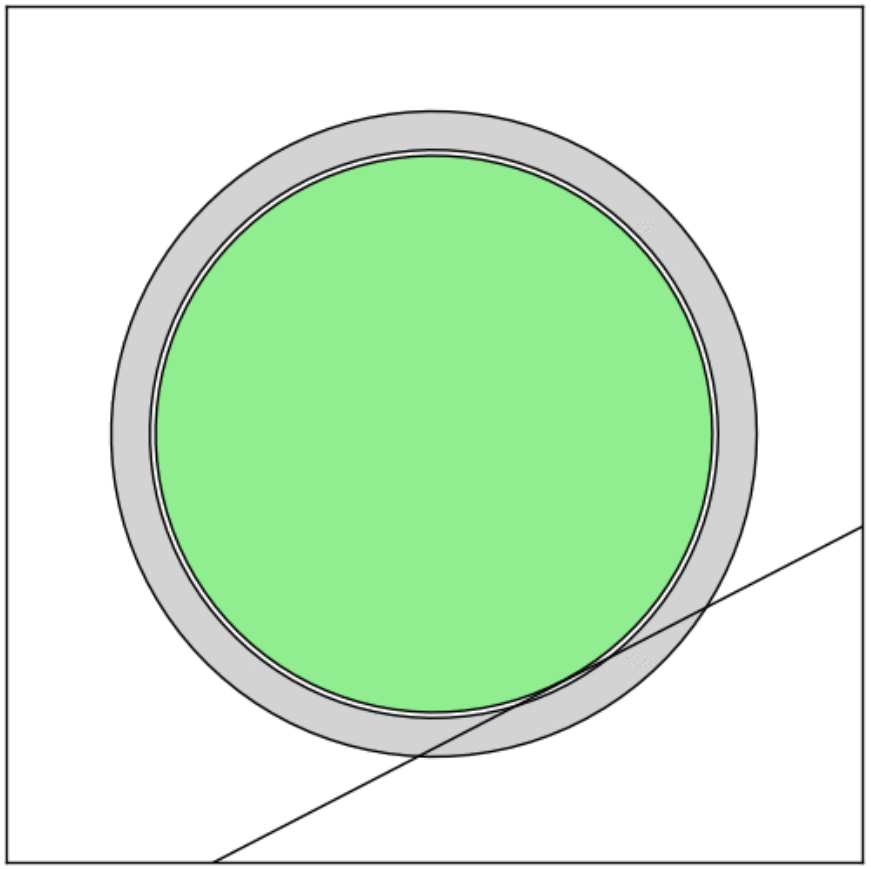


Image Source: Intel

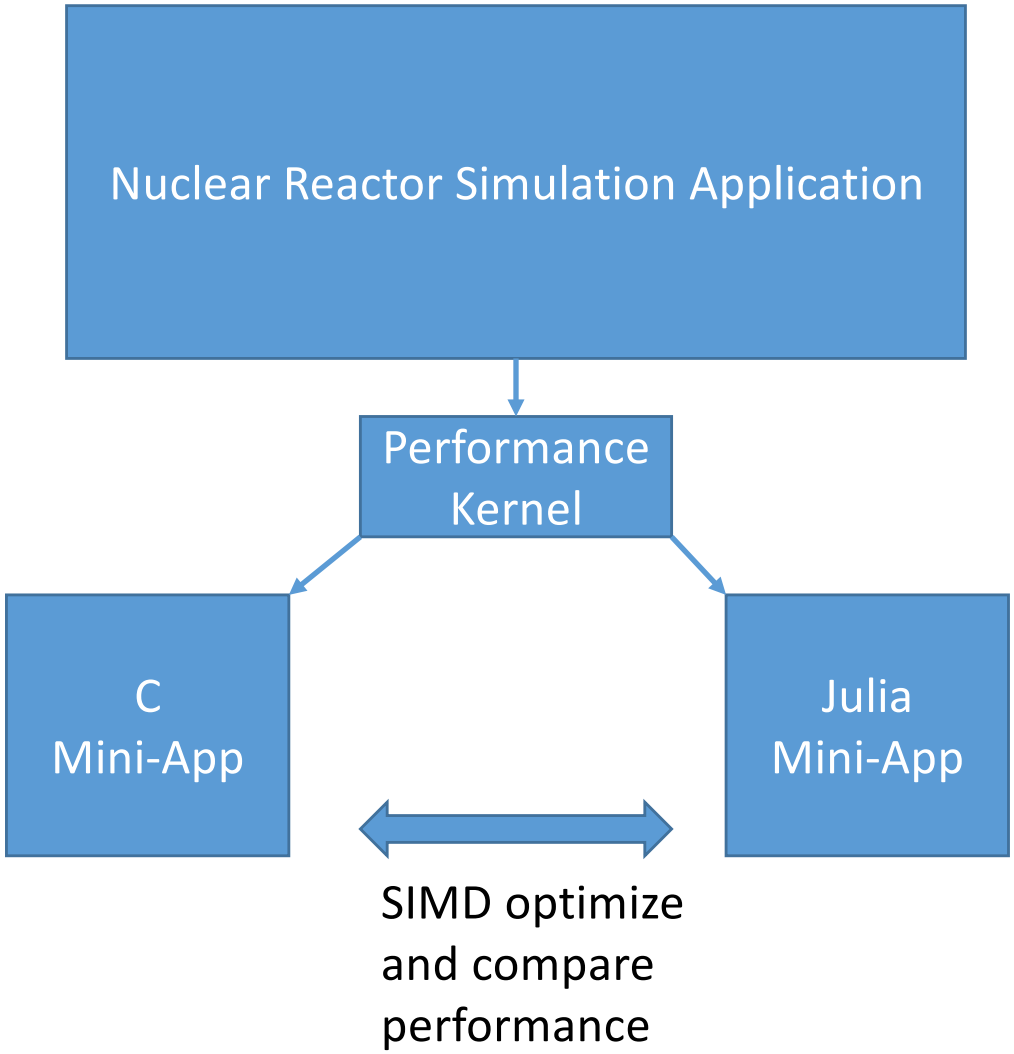
Nuclear Reactor Simulation

- Method of Characteristics (MOC)
- Not a matrix method!
- Vectorizeable inner loop

$$\psi_g(s) = \psi_g(0)e^{-\Sigma_{t_g} s} + \frac{Q_g}{\Sigma_{t_g}}\left(1 - e^{-\Sigma_{t_g} s}\right)$$



Mini-Apps Make for Easy Comparisons



Performance Kernel Pseudocode

```
for N intersections:
        Randomly sample source region
        Randomly sample material type
        Randomly sample distance d
        for each energy group g:
```

$$\Delta\psi = (\psi - Q)\left(1.0 - e^{-\Sigma d}\right)$$

$$\phi = \phi + 4\pi\Delta\psi$$

$$\psi = \psi - \Delta\psi$$

```
        end
end
```

Julia Code (Simple Version)

```
function TRRM_simple()
    # Allocate Scalar Flux Array
    scalar_flux = rand(Float64, n_source_regions * energy_groups)
    # Allocate Source Array
    source = rand(Float64, n_source_regions * energy_groups)
    # Allocate Angular Flux Vector
    angular_flux = rand(Float64, energy_groups)
    # Allocate Cross Sections
    cross_sections = rand(Float64, n_material_types * energy_groups)

    # Outer loop represents each geometrical intersection
    for i in 1:n_intersections
        # Randomly sample a source region
        source_id = rand(0:n_source_regions-1)

        # Randomly sample a material type
        material = rand(0:n_material_types-1)

        # Randomly sample a distance (cm)
        distance = rand(Float64)

        # Attenuate flux for intersection for all energy groups
        for e in 1:energy_groups
            # Compute Flux/Source index & XS index
            fs_idx = (source_id) * energy_groups + e
            xs_idx = (material) * energy_groups + e

            # Actual Computations
            tau = cross_sections[xs_idx] * distance
            q_val = source[fs_idx]
            exponential = 1.0 - exp( -tau )
            delta_psi = (angular_flux[e] - q_val) * exponential

            # Store Results
            scalar_flux[fs_idx] += 4.0 * pi * delta_psi
            angular_flux[e] -= delta_psi
        end
    end
end
```

Performance Comparison

Language	Optimization Type	Time per Integration [ns] (Lower is Better)
Julia	Unoptimized	387.79
	Basic Optimizations (types, no globals)	
	Vector SIMD	
	For Loop SIMD	
	For Loop SIMD w/Yeppp! exp()	
C	Unoptimized	
	Basic Optimizations (compiler flags)	
	SIMD Optimized	

Julia Basic Optimizations

- Explicit typing
- No global variables

Performance Comparison

Language	Optimization Type	Time per Integration [ns] (Lower is Better)
Julia	Unoptimized	387.79
	Basic Optimizations (types, no globals)	47.55
	Vector SIMD	
	For Loop SIMD	
	For Loop SIMD w/Yeppp! exp()	
C	Unoptimized	
	Basic Optimizations (compiler flags)	
	SIMD Optimized	

Julia SIMD Optimization Strategies

Simple

```
for e in 1:energy_groups
    # Computation
    # Computation
    # Computation
end
```

For Loop SIMD

```
@fastmath @inbounds @simd for e in 1:energy_groups
    # Computation
end

@fastmath @inbounds @simd for e in 1:energy_groups
    # Computation
end

@fastmath @inbounds @simd for e in 1:energy_groups
    # Computation
end
```

Vector SIMD

```
# Computations
@fastmath @inbounds A = B[i:j] .* d
# Computations
@fastmath @inbounds C = D[i:j] * A
# Computations
@fastmath @inbounds E[k:l] = A - C
```

Performance Comparison

Language	Optimization Type	Time per Integration [ns] (Lower is Better)
Julia	Unoptimized	387.79
	Basic Optimizations (types, no globals)	47.55
	Vector SIMD	47.29
	For Loop SIMD	8.73
	For Loop SIMD w/Yeppp! exp()	
C	Unoptimized	
	Basic Optimizations (compiler flags)	
	SIMD Optimized	

Yeppp! Vector Math Library

- SIMD vectorized library for Julia
- Large portion of computational time in MOC algorithm is exponential evaluation
- Call to Yeppp.exp!(exponential, -tau)

Performance Comparison

Language	Optimization Type	Time per Integration [ns] (Lower is Better)
Julia	Unoptimized	387.79
	Basic Optimizations (types, no globals)	47.55
	Vector SIMD	47.29
	For Loop SIMD	8.73
	For Loop SIMD w/Yeppp! exp()	18.37
C	Unoptimized	
	Basic Optimizations (compiler flags)	
	SIMD Optimized	

Performance Comparison

Language	Optimization Type	Time per Integration [ns] (Lower is Better)
Julia	Unoptimized	387.79
	Basic Optimizations (types, no globals)	47.55
	Vector SIMD	47.29
	For Loop SIMD	8.73
	For Loop SIMD w/Yeppp! exp()	18.37
C	Unoptimized	3.30
	Basic Optimizations (compiler flags)	
	SIMD Optimized	

Basic C Optimizations

- Unoptimized = no optimizing compiler flags
- Basic optimizations include the following intel compiler flags:
  - -fast
  - -ipo
  - -no-prec-div

Performance Comparison

Language	Optimization Type	Time per Integration [ns] (Lower is Better)
Julia	Unoptimized	387.79
	Basic Optimizations (types, no globals)	47.55
	Vector SIMD	47.29
	For Loop SIMD	8.73
	For Loop SIMD w/Yeppp! exp()	18.37
C	Unoptimized	3.30
	Basic Optimizations (compiler flags)	2.68
	SIMD Optimized	

C SIMD Optimization Strategies

Simple

```
for( int i = 0; i < energy_groups; i++ )
    // Computation
    // Computation
    // Computation
end
```

For Loop SIMD

```
#pragma omp simd
for( int i = 0; i < energy_groups; i++ )
    // Computation
    // Computation
    // Computation
end
```

Also required: aligned allocations

Performance Comparison

Language	Optimization Type	Time per Integration [ns] (Lower is Better)
Julia	Unoptimized	387.79
	Basic Optimizations (types, no globals)	47.55
	Vector SIMD	47.29
	For Loop SIMD	8.73
	For Loop SIMD w/Yeppp! exp()	18.37
C	Unoptimized	3.30
	Basic Optimizations (compiler flags)	2.68
	SIMD Optimized	1.89

Conclusions

- **SIMD in Julia v0.5**
  1. For loops are much faster than vector operations (for vector length 32)
  2. Yeppp! library did not provide speedup, as it is higher precision than what is needed by MOC algorithm
  3. Questionable if @simd works for this algorithm

- **In the context of a neutron transport simulation:**
  1. Unoptimized implementation was 117x faster in C than Julia
  2. Basic optimized implementation was 17x faster in C than Julia
  3. SIMD optimized implementation was 4.6x faster in C than Julia