From Merging Frameworks to Merging Stars: Experiences using HPX, Kokkos and SIMD Types

Gregor Daiß is a PhD student at the University of Stuttgart, specializing in high-performance computing. His main interests include task-based runtime systems, distributed computing, performance-portability as well as refactoring large-scale simulations and porting them to accelerators. Current work mostly involves both Kokkos (for performance-portability) and HPX (task-based runtime system) for these purposes.
From Merging Frameworks to Merging Stars: Experiences using HPX, Kokkos and SIMD Types

Gregor Daiß, University of Stuttgart
Scientific Application: Stellar Mergers with Octo-Tiger

Octo-Tiger

- Use-case: Simulation of binary star systems and their eventual outcomes
- Written in C++, built on the HPX runtime system
- Uses Kokkos for portable compute kernels
- Recently being used on machines like Piz Daint and Summit
- → Recent optimization work was done for GPUs
- **New target: Fugaku and the A64FX CPUs: How portable are our Kokkos kernels?**

Flow on the surface of two stars with visible mass transfer.
Scientific Application: Stellar Mergers with Octo-Tiger

Octo-Tiger

- Use-case: Simulation of binary star systems and their eventual outcomes
- Written in C++, built on the HPX runtime system
- Uses Kokkos for portable compute kernels
- Recently being used on machines like Piz Daint and Summit
- Recent optimization work was done for GPUs
- **New target:** Fugaku and the A64FX CPUs: How portable are our Kokkos kernels?

Octo-Tiger Components

- Self-gravitating astrophysical fluids on an AMR grid
- Major component 1: Gravity solver (using the fast multipole method)
- Major component 2: Hydro solver (finite volumes)
- AMR: Oct-tree with $8^3$ sub-grids in each tree node
- Each solver kernel works on one sub-grid (and its ghostlayers) at a time
- Multicore/Multinode usage by having concurrent kernel
Scientific Application: Software Dependencies (HPX + Kokkos)

HPX: A distributed task-based runtime system

Distributed:
- Unified C++ Syntax for local and remote operations
- Multiple communication backends
- Active global address space

Task-based:
- Launch functions/lambdas asynchronously as tasks.
  \[ \text{hpx::future}<\text{void}> f1 = \text{hpx::async}([]()....); \]
- Combine futures/tasks into graphs asynchronously
  \[ \text{fut.then}([]()....); \]

Runtime:
- Lightweight threads (tasks)
- Threadpool of worker threads

Kokkos: C++ Performance Portability
- Allows writing portable kernels working across a wide range of GPUs (CUDA, HIP, SYCL execution spaces)
- Allows kernel to run on CPU as well (Serial, OpenMP execution spaces)
- HPX-Kokkos integrations exist, allowing kernels to run as small HPX tasks (HPX execution space)
- HPX Futures for asynchronous kernel launches
- Many more useful abstractions (memory spaces/layouts/traits, execution policies, ...)
- Includes SIMD types for explicit vectorization inside Kokkos kernels
HPX: A distributed task-based runtime system

**Distributed:**
- Unified C++ Syntax for local and remote operations
- Multiple communication backends
- Active global address space

**Task-based:**
- Launch functions/lambdas asynchronously as tasks.
  \[
  \text{hpx::future<\text{void}> f1 = hpx::async([]()....);}
  \]
- Combine futures/tasks into graphs asynchronously
  \[
  \text{fut.then([]()....);}
  \]

**Runtime:**
- Lightweight threads (tasks)
- Threadpool of worker threads

Kokkos: C++ Performance Portability

- Allows writing portable kernels working across a wide range of GPUs (CUDA, HIP, SYCL execution spaces)
- Allows kernel to run on CPU as well (Serial, OpenMP execution spaces)
- HPX-Kokkos integrations exist, allowing kernels to run as small HPX tasks (HPX execution space)
- HPX Futures for asynchronous kernel launches
- Many more useful abstraction (memory spaces/layouts/traits, execution policies, ...)
- Includes SIMD types for explicit vectorization inside Kokkos kernels
Chaining tasks with futures

- Asynchronous kernel launches return futures
- Futures can be chained together
- Create execution graph

See [1] and [2] for more details about the HPX-Kokkos integration
Chaining tasks with futures

- Asynchronous kernel launches return futures
- Futures can be chained together
- Create execution graph

Tasks as versatile kernels using Kokkos and SIMD types

- HPX-Kokkos integrations allow us to get futures from asynchronous kernel launches

**Execution Model**

### Chaining tasks with futures

- Asynchronous kernel launches return futures
- Futures can be chained together
- Create execution graph

### Tasks as versatile kernels using Kokkos and SIMD types

- HPX-Kokkos integrations allow us to get futures from asynchronous kernel launches
- Kokkos HPX execution space allows us to run Kokkos kernels on HPX worker threads
- SIMD Types allow us to use explicit vectorization inside a kernel while maintaining GPU support
## Execution Model

### Chaining tasks with futures
- Asynchronous kernel launches return futures
- Futures can be chained together
- Create execution graph

### Tasks as versatile kernels using Kokkos and SIMD types
- HPX-Kokkos integrations allow us to get futures from asynchronous kernel launches
- Kokkos HPX execution space allows us to run Kokkos kernels on HPX worker threads
- SIMD Types allow us to use explicit vectorization inside a kernel while maintaining GPU support

### Diagram

<table>
<thead>
<tr>
<th>Launch Kernel asynchronously</th>
<th>Split Kernel into Compute blocks via Kokkos</th>
<th>SIMD Types Parallellization</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPX Application (Octo-Tiger)</td>
<td>HPX-Kokkos executor</td>
<td>GPU Execution</td>
</tr>
<tr>
<td>Launch Kokkos Kernel from arbitrary thread</td>
<td>Receive HPX future that will be ready when the kernel is finished</td>
<td>CUDA Execution Space</td>
</tr>
<tr>
<td>CPU Execution</td>
<td>Task 1 ... Task N</td>
<td>SIMD Types</td>
</tr>
<tr>
<td>Task 1 ... Task N</td>
<td></td>
<td>Use scalar instantiation for GPU</td>
</tr>
</tbody>
</table>

**What is missing?**

---

From Merging Frameworks to Merging Stars: Experiences using HPX, Kokkos and SIMD Types  
Gregor Dais
Additions / Adaptations for A64FX

Addition 1: Add SIMD Types / Masks to Hydro Kernels

1. Explicit SIMD vectorization needs to be added to the kernels in question
2. Hydro kernels required a lot of masking due to branching in the Reconstruct kernel
3. Requires manual effort but guarantees SIMD usage across all platforms (supported by the types)

Work in Progress: Octo-Tiger SIMD Support (Before)

<table>
<thead>
<tr>
<th>Kokkos Kernels: Gravity Solver</th>
<th>Kokkos Kernels: Hydro Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multipole Rho</td>
<td>SIMD Support</td>
</tr>
<tr>
<td>Multipole</td>
<td>SIMD Support</td>
</tr>
<tr>
<td>Monopole</td>
<td>SIMD Support</td>
</tr>
<tr>
<td>Multipole Root Rho</td>
<td>SIMD Support</td>
</tr>
<tr>
<td>Multipole Root</td>
<td>SIMD Support</td>
</tr>
<tr>
<td>Reconstruct</td>
<td>No SIMD Support</td>
</tr>
<tr>
<td>Flux</td>
<td>No SIMD Support</td>
</tr>
</tbody>
</table>
Addition 1: Add SIMD Types / Masks to Hydro Kernels

1. Explicit SIMD vectorization needs to be added to the kernels in question.
2. Hydro kernels required a lot of masking due to branching in the Reconstruct kernel.
3. Requires manual effort but guarantees SIMD usage across all platforms (supported by the types).

<table>
<thead>
<tr>
<th>Work in Progress: Octo-Tiger SIMD Support (Current)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kokkos Kernels: Gravity Solver</strong></td>
</tr>
<tr>
<td>Multipole Rho</td>
</tr>
<tr>
<td>Multipole</td>
</tr>
<tr>
<td>Monopole</td>
</tr>
<tr>
<td>Multipole Root Rho</td>
</tr>
<tr>
<td>Multipole Root</td>
</tr>
<tr>
<td><strong>Kokkos Kernels: Hydro Solver</strong></td>
</tr>
<tr>
<td>Reconstruct</td>
</tr>
<tr>
<td>Flux</td>
</tr>
</tbody>
</table>

From Merging Frameworks to Merging Stars: Experiences using HPX, Kokkos and SIMD Types

Gregor Daiß

6 / 15
Additions / Adaptations for A64FX

Addition 1: Add SIMD Types / Masks to Hydro Kernels

1. Explicit SIMD vectorization needs to be added to the kernels in question
2. Hydro kernels required a lot of masking due to branching in the Reconstruct kernel
3. Requires manual effort but guarantees SIMD usage across all platforms (supported by the types)

Optimization for Small Kernels

1. When a Kokkos kernel is small enough to be run as one task, it will execute immediately on the calling thread
2. Steered via chunk size parameter at launch time
Additions / Adaptations for A64FX

Addition 2: Integrate `std::experimental::simd (SES)` types

1. Interface is already similar for both the types and the masks
2. Add missing functionality and map the functions that have different names (`where`)
3. Namespace change
4. Instantiate kernels with SES types
Additions / Adaptations for A64FX

Addition 2: Integrate `std::experimental::simd` (SES) types

1. Interface is already similar for both the types and the masks
2. Add missing functionality and map the functions that have different names (`where`)
3. Namespace change
4. Instantiate kernels with SES types
Additions / Adaptations for A64FX

Addition 2: Integrate `std::experimental::simd (SES)` types

1. Interface is already similar for both the types and the masks
2. Add missing functionality and map the functions that have different names (`where`)
3. Namespace change
4. Instantiate kernels with SES types

---

**Available SIMD Types (Current)**

<table>
<thead>
<tr>
<th>CPU Execution Space</th>
<th>GPU Execution Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kokkos SIMD</td>
<td>std::experimental::simd</td>
</tr>
<tr>
<td>Scalar</td>
<td>Scalar</td>
</tr>
<tr>
<td>NEON</td>
<td>NEON</td>
</tr>
<tr>
<td>AVX</td>
<td>AVX</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>AVX512</td>
<td>AVX512</td>
</tr>
<tr>
<td>SVE?</td>
<td>SVE?</td>
</tr>
</tbody>
</table>

---

Kokkos Kernel | SIMD Support
---

From Merging Frameworks to Merging Stars: Experiences using HPX, Kokkos and SIMD Types

Gregor Daiß
Additions / Adaptations for A64FX

Addition 2: Integrate `std::experimental::simd` (SES) types

1. Interface is already similar for both the types and the masks
2. Add missing functionality and map the functions that have different names (where)
3. Namespace change
4. Instantiate kernels with SES types

Addition 3: Add SES-compatible SVE types

1. Use fixed length with 512 bits
2. Directly map operations to SVE functions
4. In case of unimplemented function: Serial fallback (pow)
5. Compatible with `std::experimental::simd`
# Test Setup

## Hardware

- A two-socket AMD® EPYC™ 7H12 CPU @ 2.60GHz with 128 cores
- A two-socket Intel® Xeon® Platinum 8358 CPU @ 2.60GHz with 64 cores (Intel® Icelake node)
- A Fujitsu A64FX™ CPU @ 1.80 GHz on Stony Brook University’s Ookami cluster with 48 cores (ARM® node)

## Octo-Tiger Scenario 1

- Rotating Star Scenario: Simulating a rotating star in equilibrium
- Includes all solvers and additional work (tree traversals, determination of time-step size, ...)
- AMR is turned off: Scenario uses 512 leaf sub-grids (with 512 cells each)

## Octo-Tiger Scenario 2

- Sedov Blast Wave: Hydro-only test scenario
- Scenario uses 512 leaf sub-grids (with 512 cells each)
Test 1: Single Core SIMD Speedup

- Analyze single core SIMD speedup using both the Kokkos SIMD types and the `std::experimental::simd` types
- Multiple runs (Octo-Tiger needs to be recompiled in between using different types)
- Using scenario 1 (rotating star)
- Auto-vectorization not disabled for runs with scalar types
- Gets us the overall application speedup and (using APEX) the individual kernel speedups
Test 1: Single Core SIMD Speedup

AVX on Epyc

Hydro + Gravity Scenario on AMD EPYC 7H12: Component SIMD Speedups

Application Speedup:
2.7 (Kokkos SIMD), 2.4 (SES)
Test 1: Single Core SIMD Speedup

### AVX on Epyc

<table>
<thead>
<tr>
<th>Component</th>
<th>Speedup w.r.t. KOKKOS SCALAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire Application</td>
<td>3.0</td>
</tr>
<tr>
<td>Multipole Rho</td>
<td>3.5</td>
</tr>
<tr>
<td>Reconstruct</td>
<td>3.2</td>
</tr>
<tr>
<td>Multipole</td>
<td>3.1</td>
</tr>
<tr>
<td>Monopole</td>
<td>2.9</td>
</tr>
<tr>
<td>Flux</td>
<td>2.7</td>
</tr>
<tr>
<td>Root Multipole Rho</td>
<td>2.5</td>
</tr>
<tr>
<td>Root Multipole</td>
<td>2.4</td>
</tr>
</tbody>
</table>

Application Speedup: 2.7 (Kokkos SIMD), 2.4 (SES)

### AVX512 on Intel

<table>
<thead>
<tr>
<th>Component</th>
<th>Speedup w.r.t. KOKKOS SCALAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire Application</td>
<td>5.0</td>
</tr>
<tr>
<td>Multipole Rho</td>
<td>6.0</td>
</tr>
<tr>
<td>Reconstruct</td>
<td>5.5</td>
</tr>
<tr>
<td>Multipole</td>
<td>5.2</td>
</tr>
<tr>
<td>Monopole</td>
<td>5.0</td>
</tr>
<tr>
<td>Flux</td>
<td>4.8</td>
</tr>
<tr>
<td>Root Multipole Rho</td>
<td>4.5</td>
</tr>
<tr>
<td>Root Multipole</td>
<td>4.3</td>
</tr>
</tbody>
</table>

Application Speedup: 2.5 (Kokkos SIMD), 2.3 (SES)
Scientific Application: Stellar Mergers

Execution Model

Additions / Adaptations for A64FX

Performance Results

Conclusion

Test 1: Single Core SIMD Speedup

AVX on Epyc

Hydro + Gravity Scenario on AMD EPYC 7H12: Component SIMD Speedups

AVX512 on Intel

Hydro + Gravity Scenario on Intel Xeon Platinum 8358: Component SIMD Speedups

SVE on A64FX

Hydro + Gravity Scenario on A64FX: Component SIMD Speedups

Application Speedup:

2.7 (Kokkos SIMD), 2.4 (SES)

Application Speedup:

2.5 (Kokkos SIMD), 2.3 (SES)

Application Speedup:

2.1 (SES)
Test 2

- Using scenario 1 (rotating star)
- Scale from single core to all available cores
- One run with the best SIMD extensions found for the platform in test 1
- Second run with scalar types for comparison
- Plot runtime and parallel efficiency for both runs on each platform
- Vary number of tasks used for each kernel invocation (usually just one)
Test 2: Node-Level Scaling

Runtime/Efficiency Overview

- Fujitsu A64FX (with SES)
  - Best runtime (48 cores): 39.72s
  - Best runtime with SVE (48 cores): 20.67s
  - Parallel efficiency at 48 cores with SVE: 67%

SVE on A64FX

Hydro + Gravity Scenario on A64FX: Node-Level Scaling with SES SIMD

- Computation Time [Scalar]
- Computation Time [SVE]
- Parallel Efficiency [Scalar]
- Parallel Efficiency [SVE]
Test 2: Node-Level Scaling

Runtime/Efficiency Overview

- Fujitsu A64FX (with SES)
  - Best runtime (48 cores): 39.72s
  - Best runtime with SVE (48 cores): 20.67s
  - Parallel efficiency at 48 cores with SVE: 67%
- Intel Xeon Platinum 8358 (with Kokkos SIMD)
  - Best runtime (64 cores): 6.54s
  - Best runtime with AVX512 (64 cores): 3.65s
  - Parallel efficiency with AVX512: 39%
Test 2: Node-Level Scaling

Runtime/Efficiency Overview

- **Fujitsu A64FX (with SES)**
  - Best runtime (48 cores): 39.72s
  - Best runtime with SVE (48 cores): 20.67s
  - Parallel efficiency at 48 cores with SVE: 67%
- **Intel Xeon Platinum 8358 (with Kokkos SIMD)**
  - Best runtime (64 cores): 6.54s
  - Best runtime with AVX512 (64 cores): 3.65s
  - Parallel efficiency with AVX512: 39%
- **AMD EPYC 7H12 CPU (with Kokkos SIMD)**
  - Best runtime (64 cores): 7.97s
  - Best runtime with AVX (64 cores): 4.78s
  - Parallel efficiency with AVX: 31%

AVX on Epyc

Hydro + Gravity Scenario on AMD EPYC 7H12: Node-Level Scaling with KOKKOS SIMD

- Computation Time [Scalar]
- Computation Time [AVX]
- Parallel Efficiency [Scalar]
- Parallel Efficiency [AVX]
Test 2: Node-Level Scaling – Different Task Setup

Changing Multipole kernel launch configuration from one task to 16 tasks:

A64FX: One task per Multipole kernel

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Computation Time [Scalar]</th>
<th>Computation Time [SVE]</th>
<th>Parallel Efficiency [Scalar]</th>
<th>Parallel Efficiency [SVE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1357.25</td>
<td>667.89</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>667.89</td>
<td>39.72</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>39.72</td>
<td>20.67</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>8</td>
<td>20.67</td>
<td>13.91</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>16</td>
<td>13.91</td>
<td>8.72</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>32</td>
<td>8.72</td>
<td>5.91</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>48</td>
<td>5.91</td>
<td>4.57</td>
<td>40</td>
<td>40</td>
</tr>
</tbody>
</table>

A64FX: 16 tasks per Multipole kernel

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Computation Time [Scalar]</th>
<th>Computation Time [SVE]</th>
<th>Parallel Efficiency [Scalar]</th>
<th>Parallel Efficiency [SVE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1370.42</td>
<td>676.12</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>676.12</td>
<td>31.91</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>31.91</td>
<td>17.01</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>8</td>
<td>17.01</td>
<td>10.12</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>16</td>
<td>10.12</td>
<td>6.67</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>32</td>
<td>6.67</td>
<td>4.72</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>48</td>
<td>4.72</td>
<td>3.67</td>
<td>40</td>
<td>40</td>
</tr>
</tbody>
</table>

→ Runtime improves from 20.67s to 17.01s.
Changing Multipole kernel launch configuration from one task to 16 tasks:

→ Runtime improves from 3.65s to 3.3s.
Node-Level Scaling and Parallel Efficiency

Test 3: Configuration / Results

- Using scenario 2 (Sedov Blast Wave).
- Hydro-only scenario for runtimes of the new hydro SIMD implementations
- Application Speedup w.r.t Kokkos SCALAR (Single Core)
  - Intel: 2.1
  - AMD: 1.8
  - A64FX: 2.3

On a single core

Hydro-Only Scenario using one CPU Core

Hydro Computation Time in s

<table>
<thead>
<tr>
<th>Compute Node</th>
<th>Intel Icelake</th>
<th>AMD Epyc</th>
<th>A64FX</th>
</tr>
</thead>
<tbody>
<tr>
<td>91.97</td>
<td>95.95</td>
<td>114.54</td>
<td>970.04</td>
</tr>
<tr>
<td>94.95</td>
<td>45.73</td>
<td>62.5</td>
<td>881.65</td>
</tr>
<tr>
<td>95.73</td>
<td>114.54</td>
<td>389.33</td>
<td>970.04</td>
</tr>
</tbody>
</table>

From Merging Frameworks to Merging Stars: Experiences using HPX, Kokkos and SIMD Types

Gregor Daiß
Test 3: Configuration / Results

- Using scenario 2 (Sedov Blast Wave).
- Hydro-only scenario for runtimes of the new hydro SIMD implementations
- Application Speedup w.r.t Kokkos SCALAR (Single Core)
  - Intel: 2.1
  - AMD: 1.8
  - A64FX: 2.3

Using all cores

Hydro-Only Scenario using 64 Cores (48 on A64FX)
Conclusion and Outlooks

Conclusion

- Contributions of this work:
  - Added SIMD support for the Octo-Tiger Kokkos hydro kernel
  - Integration of `std::experimental::simd` types with Kokkos kernels
  - Addition of `std::experimental::simd-compatible` SVE SIMD types
- Our changes substantially reduced the Octo-Tiger runtime on A64FX (and also yielded noticeable benefits on other CPUs)
- Our new SVE SIMD types work well (and can of course be used in other applications)
Conclusion and Outlooks

Conclusion

- Contributions of this work:
  - Added SIMD support for the Octo-Tiger Kokkos hydro kernel
  - Integration of std::experimental::simd types with Kokkos kernels
  - Addition of std::experimental::simd-compatible SVE SIMD types
- Our changes substantially reduced the Octo-Tiger runtime on A64FX (and also yielded noticeable benefits on other CPUs)
- Our new SVE SIMD types work well (and can of course be used in other applications)

Application-Independent Execution Model:

- HPX allows us to easily express dependencies between kernels (and other parts of the application)
- Kokkos kernels allow us to easily use HPX worker threads for CPU execution (whilst maintaining GPU support)
- Explicit vectorization within the kernels with SIMD types works robustly across multiple CPU architectures/compilers
Conclusion and Outlooks

Conclusion

- Contributions of this work:
  - Added SIMD support for the Octo-Tiger Kokkos hydro kernel
  - Integration of std::experimental::simd types with Kokkos kernels
  - Addition of std::experimental::simd-compatible SVE SIMD types

- Our changes substantially reduced the Octo-Tiger runtime on A64FX (and also yielded noticeable benefits on other CPUs)

- Our new SVE SIMD types work well (and can of course be used in other applications)

Application-Independent Execution Model:

- HPX allows us to easily express dependencies between kernels (and other parts of the application)
- Kokkos kernels allow us to easily use HPX worker threads for CPU execution (whilst maintaining GPU support)
- Explicit vectorization within the kernels with SIMD types works robustly across multiple CPU architectures/compilers

Outlook

- Investigate node-level scaling on Epyc
- Optimize the first SIMD implementation of the kernels
- Optimize runtime on A64FX in general
- Further investigate different launch configurations (split kernels into more tasks), especially for distributed scenarios
Conclusion and Outlooks

Conclusion

• Contributions of this work:
  • Added SIMD support for the Octo-Tiger Kokkos hydro kernel
  • Integration of `std::experimental::simd` types with Kokkos kernels
  • Addition of `std::experimental::simd`-compatible SVE SIMD types

• Our changes substantially reduced the Octo-Tiger runtime on A64FX (and also yielded noticeable benefits on other CPUs)

• Our new SVE SIMD types work well (and can of course be used in other applications)

Application-Independent Execution Model:

• HPX allows us to easily express dependencies between kernels (and other parts of the application)

• Kokkos kernels allow us to easily use HPX worker threads for CPU execution (whilst maintaining GPU support)

• Explicit vectorization within the kernels with SIMD types works robustly across multiple CPU architectures/compilers

Outlook

• Investigate node-level scaling on Epyc

• Optimize the first SIMD implementation of the kernels

• Optimize runtime on A64FX in general

• Further investigate different launch configurations (split kernels into more tasks), especially for distributed scenarios

Acknowledgment

The authors would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the innovative high-performance Ookami computing system, which was made possible by a $5M National Science Foundation grant (#1927880).
Thank you for your attention!
G. Daiß et al. (video presentation) hips 2021: Beyond fork-join: Integration of performance portable kokkos kernels with hpx. Youtube. [Online]. Available: https://www.youtube.com/watch?v=CQaA9AYlm1I