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

Gregor Daiß, PhD Student, University of Stuttgart, Germany

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.

SC22
Dallas, TX | hpc accelerates.

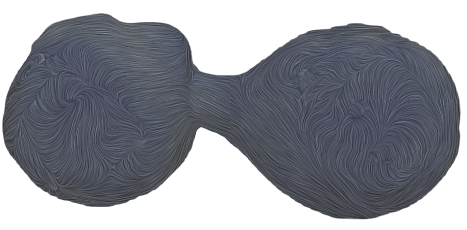
From Merging Frameworks to Merging Stars: Experiences using HPX, Kokkos and SIMD Types

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.

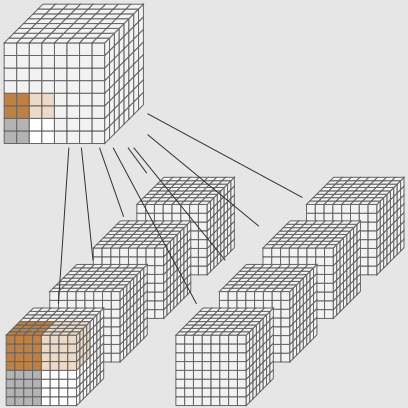
Scientific Application: Stellar Mergers with Octo-Tiger

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 asynchrounosely as tasks.
  `hpx::future<void> f1 = hpx::async([]().....);`
- Combine futures/tasks into graphs asynchrounosely
  `fut.then([]().....);`

Runtime:

- Lightweight threads (tasks)
- Threadpool of worker threads

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 asynchrounosely as tasks.
  `hpx::future<void> f1 = hpx::async([]()....);`
- Combine futures/tasks into graphs asynchrounosely
  `fut.then([]()....);`

Runtime:

- Lightweight threads (tasks)
- Threadpool of worker threads

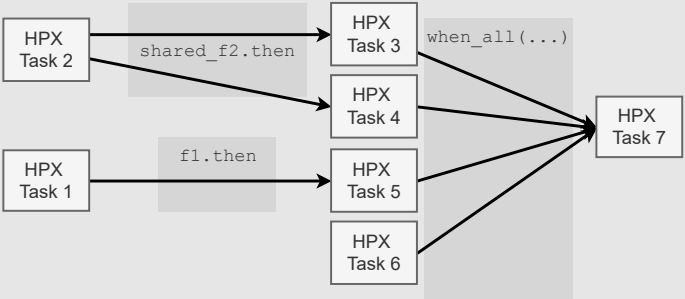
Kokkos: C++ Performance Portability

- 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

Execution Model

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

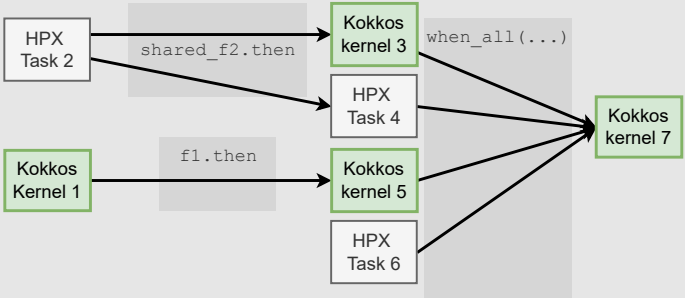
Execution Model

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



See [1] and [2] for more details about the HPX-Kokkos integration

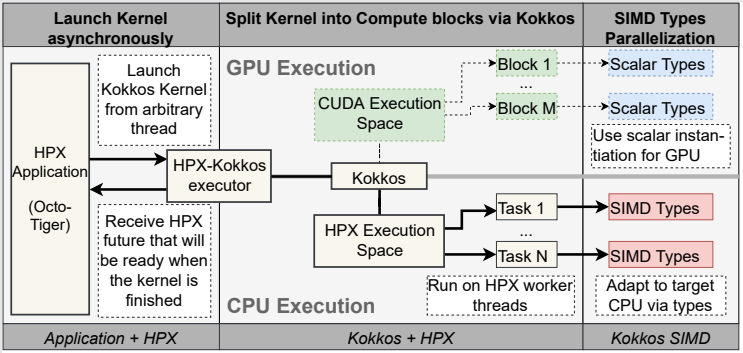
Execution Model

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



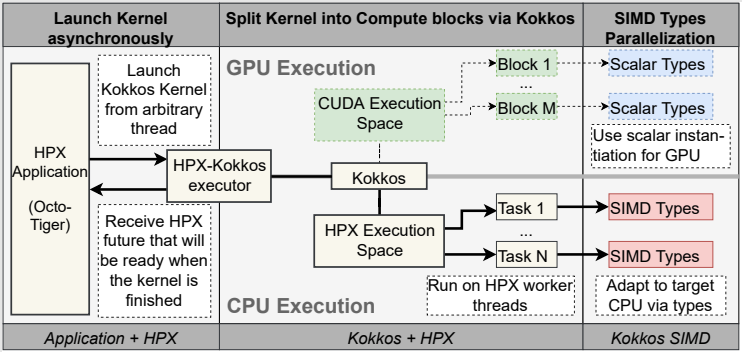
Execution Model

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



What is missing?

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)

Kokkos Kernels: Gravity Solver	
Multipole Rho	SIMD Support
Multipole	SIMD Support
Monopole	SIMD Support
Multipole Root Rho	SIMD Support
Multipole Root	SIMD Support

Kokkos Kernels: Hydro Solver	
Reconstruct	No SIMD Support
Flux	No SIMD Support

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 (Current)

Kokkos Kernels: Gravity Solver	
Multipole Rho	SIMD Support
Multipole	SIMD Support
Monopole	SIMD Support
Multipole Root Rho	SIMD Support
Multipole Root	SIMD Support

Kokkos Kernels: Hydro Solver	
Reconstruct	SIMD Support
Flux	SIMD Support

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

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

Kokkos Kernels: Gravity Solver	
Multipole Rho	SIMD Support
Multipole	SIMD Support
Monopole	SIMD Support
Multipole Root Rho	SIMD Support
Multipole Root	SIMD Support

Kokkos Kernels: Hydro Solver	
Reconstruct	SIMD Support
Flux	SIMD Support

Additions / Adaptations for A64FX

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

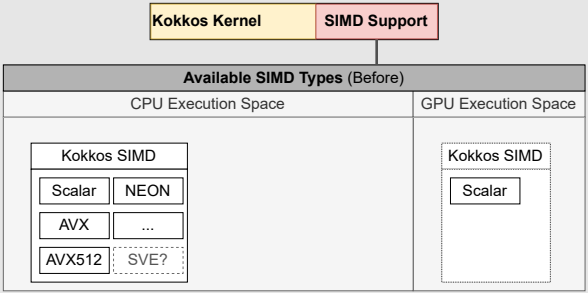
1. Interface is already similar for both the types and the masks
2. Add missing functionality and map the functions that have different names (`where`)
3. Namespace change
4. Instantiate kernels with SES types

Kokkos Kernel	SIMD Support

Additions / Adaptations for A64FX

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

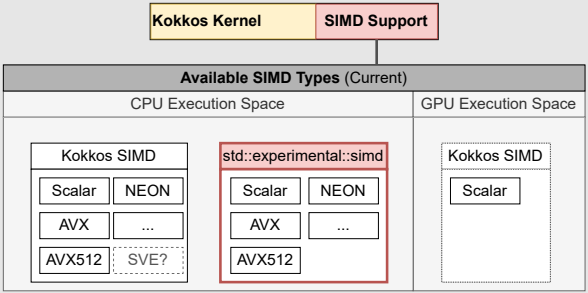
1. Interface is already similar for both the types and the masks
2. Add missing functionality and map the functions that have different names (`where`)
3. Namespace change
4. Instantiate kernels with SES types



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



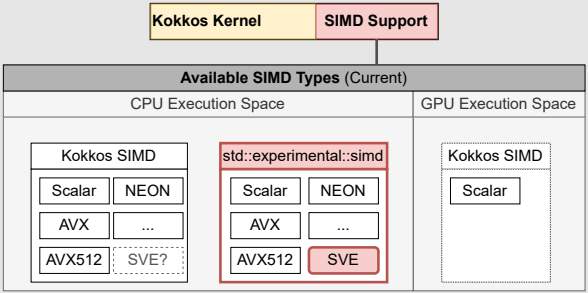
Additions / Adaptations for A64FX

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

Addition 3: Add SES-compatible SVE types

1. Use fixed length with 512 bits
2. Directly map operations to SVE functions
3. Support masking via types.
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)

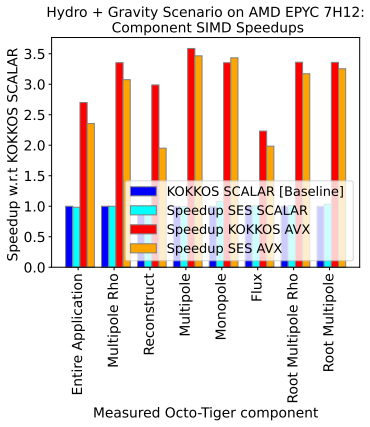
Test 1: Single Core SIMD Speedup

Test 1

- 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)
- Autovectorization 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

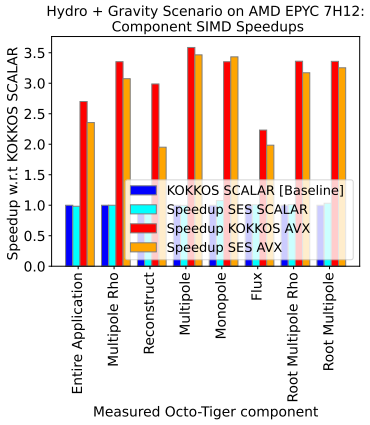
AVX on Epyc



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

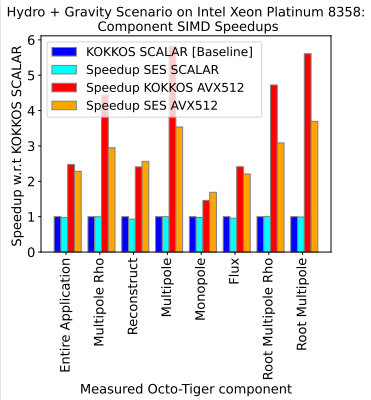
Test 1: Single Core SIMD Speedup

AVX on Epyc



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

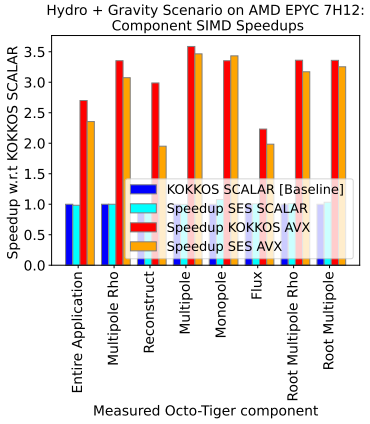
AVX512 on Intel



Application Speedup:
2.5 (Kokkos SIMD), 2.3 (SES)

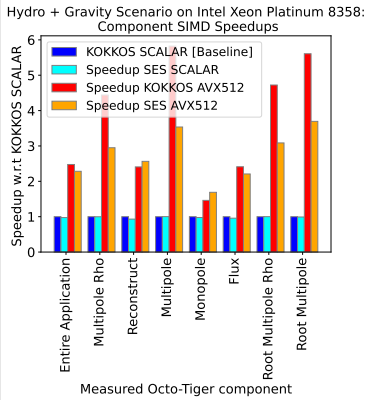
Test 1: Single Core SIMD Speedup

AVX on Epyc



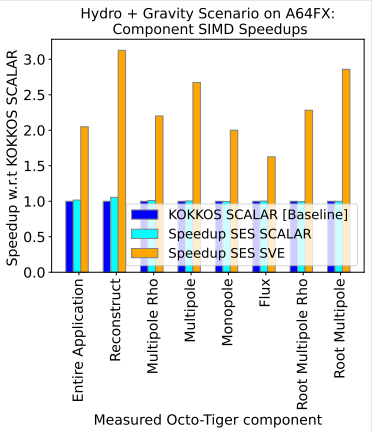
Application Speedup:
2.7 (Kokkos SIMD), 2.4 (SES)

AVX512 on Intel



Application Speedup:
2.5 (Kokkos SIMD), 2.3 (SES)

SVE on A64FX



Application Speedup:
2.1 (SES)

Node-Level Scaling and Parallel Efficiency

Test 2

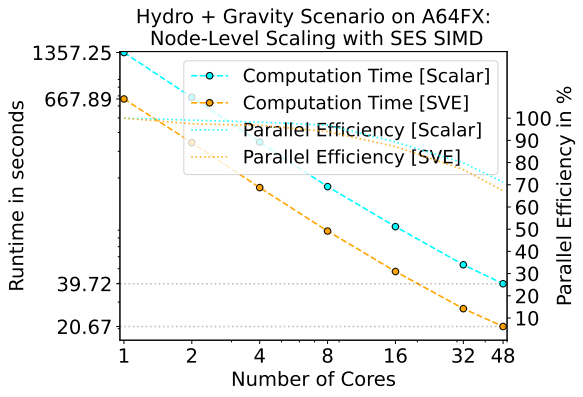
- Using scenario 1 (rotating star)
- Scale from single core to all available cores
- One run with the best SIMD extensions found for the platform in test 1
- Second run with scalar types for comparison
- Plot runtime and parallel efficiency for both runs on each platform
- Vary number of tasks used for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

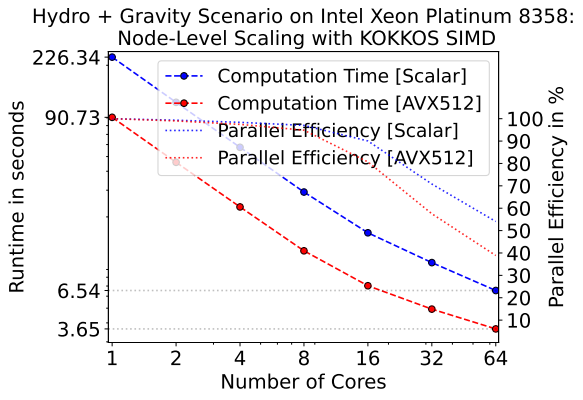


Test 2: Node-Level Scaling

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%

AVX512 on Intel

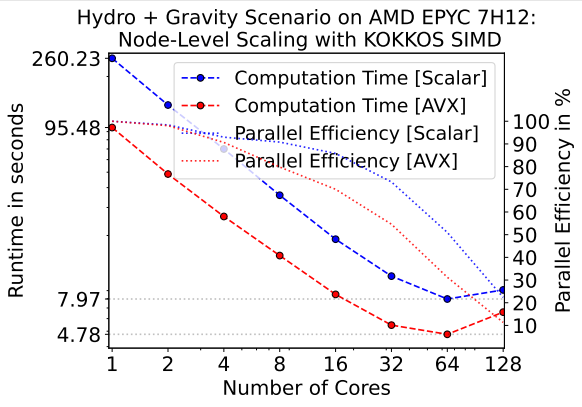


Test 2: Node-Level Scaling

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%

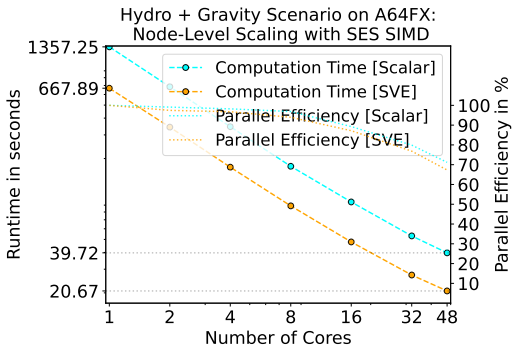
AVX on Epyc



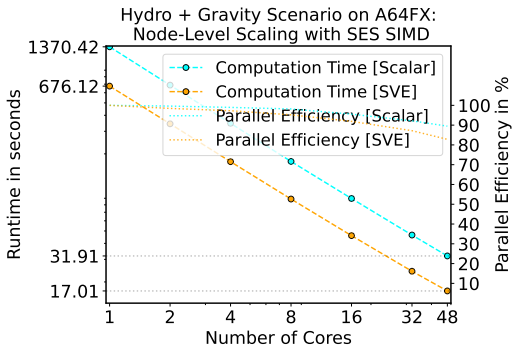
Test 2: Node-Level Scaling – Different Task Setup

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

A64FX: One task per Multipole kernel



A64FX: 16 tasks per Multipole kernel

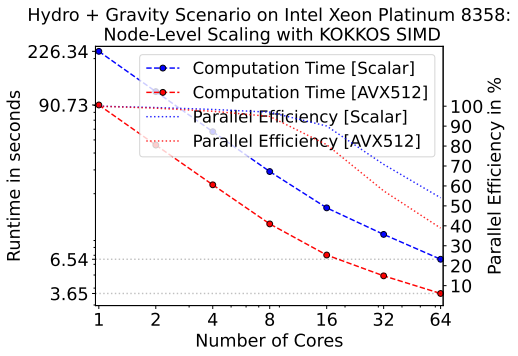


→ Runtime improves from 20.67s to 17.01s.

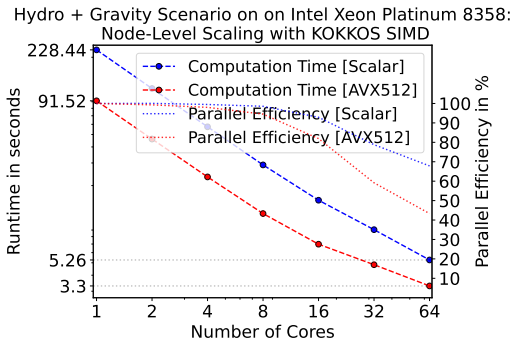
Test 2: Node-Level Scaling – Different Task Setup

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

Intel: One task per Multipole kernel



Intel: 16 tasks per Multipole kernel



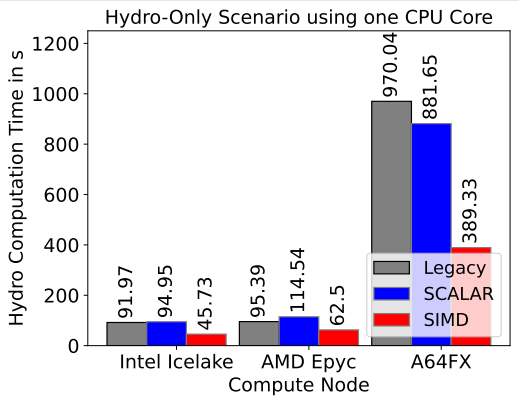
→ Runtime improves from 3.65s to 3.3s.

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

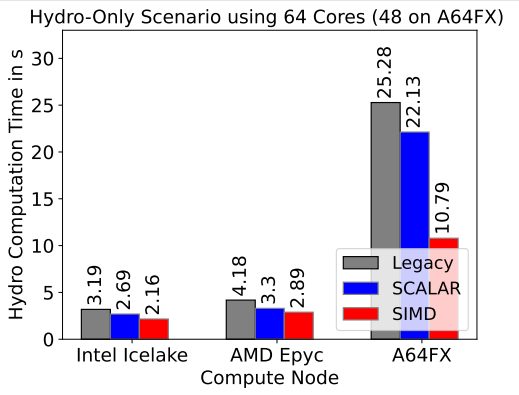


Node-Level Scaling and Parallel Efficiency

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

Using all cores



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)

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

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)

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

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

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).

Universität Stuttgart

LSU Center for Computation & Technology
Interdisciplinary | Innovative | Inventive

STE||AR GROUP

Thank you for your attention!

SC22
Dallas, TX | hpc accelerates.

IPVS

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

——, “Beyond fork-join: Integration of performance portable Kokkos kernels with HPX,” in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 377–386.