FAST. FLEXIBLE. FREE.
GROMACS

oneAPI Case Study: GROMACS

Andrey Alekseenko

KTH Royal Institute of Technology & SciLifeLab
Stockholm, Sweden

GROMACS

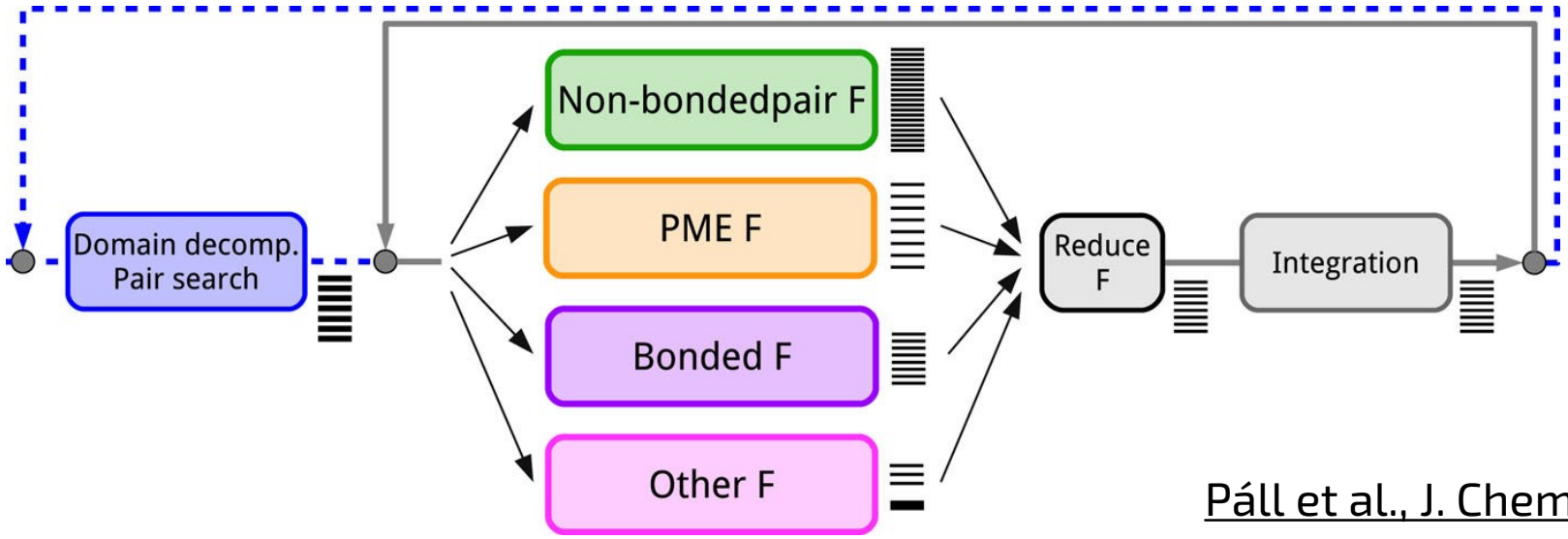
- Open-source molecular dynamics engine
- One of the most used HPC codes worldwide
- High-performance for a wide range of modeled systems
- … and on a wide range of platforms:
  - from supercomputers to laptops (Folding@Home)
  - x86, x86-64, ARM, POWER, SPARC, RISC-V
  - 11 SIMD backends
  - AMD, Apple, Intel, and NVIDIA GPUs; Intel Xeon Phi
  - Windows, MacOS, BSD, included in many Linux distros

GROMACS 2024 (upcoming)

- (Mostly) modern C++17 codebase
  - 468k lines of C++ code
  - With a bit of legacy (first release: 1991)
- MPI for inter-node parallelism
- OpenMP for multithreading
- SIMD for low-latency operations on CPU
- GPU offload for high-throughput operations
  - CUDA: NVIDIA
  - OpenCL: AMD, Apple, Intel, NVIDIA
  - SYCL: AMD, Intel, NVIDIA

Molecular dynamics

- Iterative problem
  - Like N-body, but with fancier physics
- One step ~1 fs, need to simulate µs to ms
  - 10^9-10^{12} steps



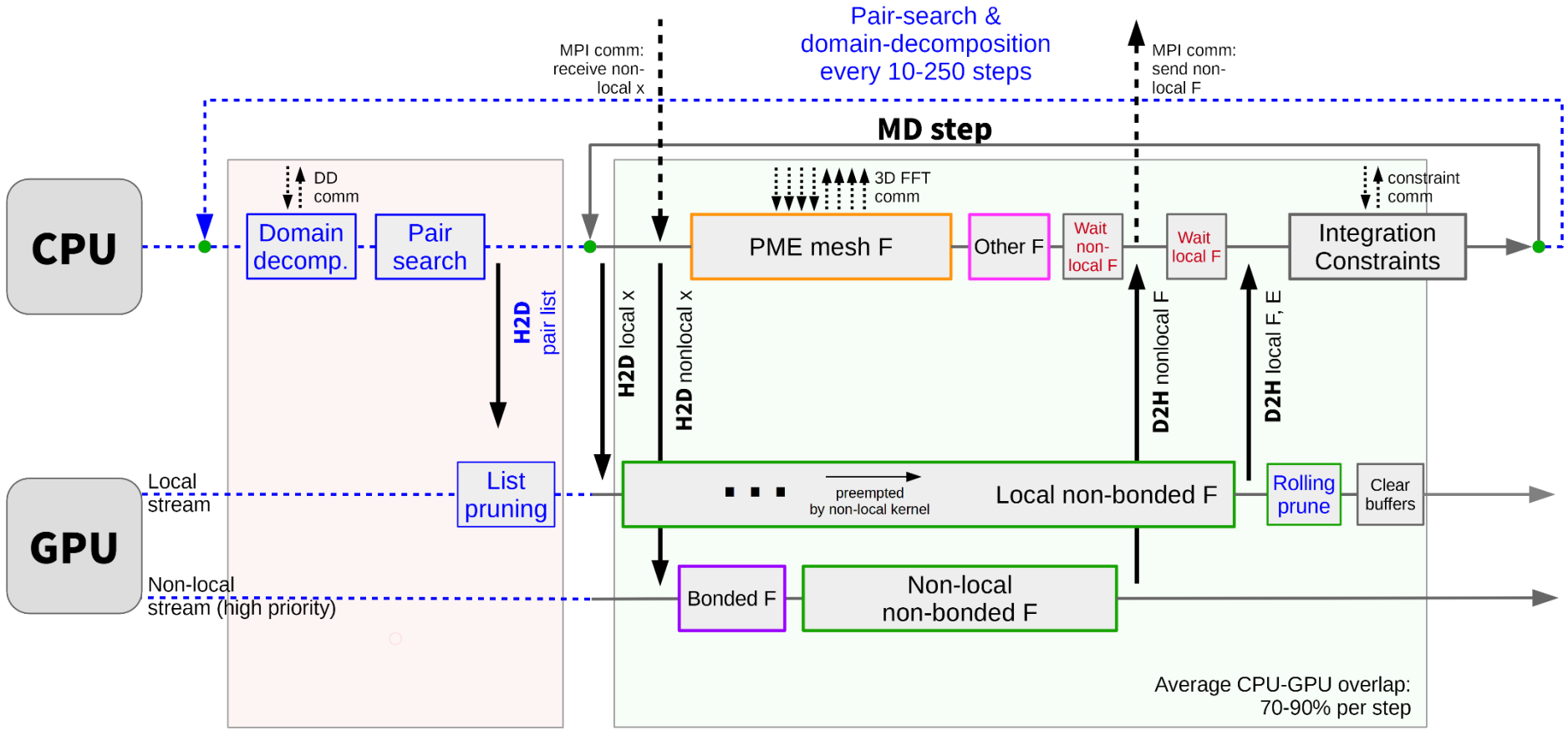
Páll et al., J. Chem. Phys. 153, 134110 (2020)

Heterogeneous parallelization

- Minimize latency
- Minimize CPU and GPU stalls
- Minimize data exchange between host and device
  - And between nodes

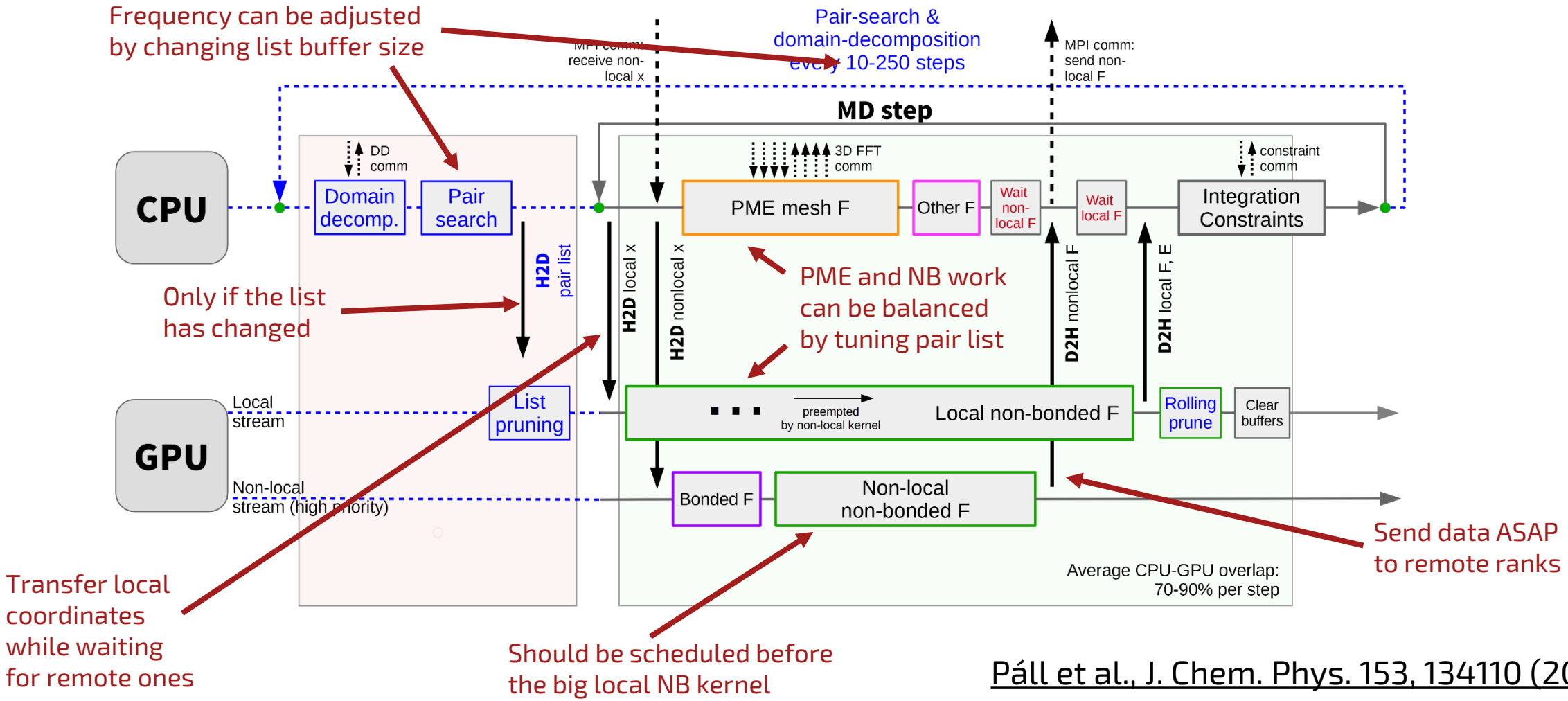
- Optimal offloading scheme depends on simulated system
  - And on available hardware
- Must be maintainable

Molecular dynamics: real schedule



Páll et al., J. Chem. Phys. 153, 134110 (2020)

Molecular dynamics: real schedule



Páll et al., J. Chem. Phys. 153, 134110 (2020)

GPU feature support in GROMACS 2020

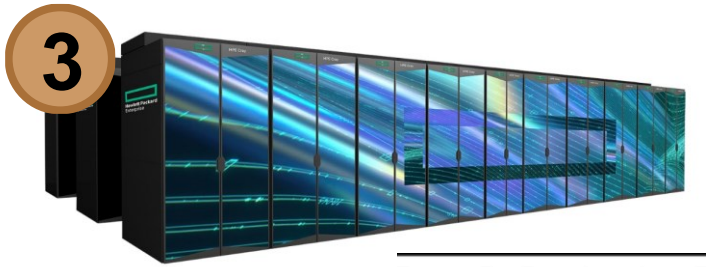
	NVIDIA CUDA	OpenCL
Non-bonded offload	√	√
PME offload	√	√
Update offload	√	X
Bonded offload	√	X
Direct GPU-GPU comm	√	X
Hardware support	NVIDIA	NVIDIA, AMD, Intel

Why another GPU framework?

AMD Instinct GPU

Intel Data Center GPU Max

First exascale systems



LUMI

Why not OpenCL?

- OpenCL kernels are C99, the rest of GROMACS is C++17
  - C++ kernels are not widely supported
  - Separate-source model
  - Hard to maintain
- Supported by all vendors, preferred by none
- No interoperability with native libraries
  - GPU-aware MPI is a must on modern systems

Why SYCL?

- Open standard, free (libre) implementations
- Implemented on top of existing backends
  - Intel® oneAPI DPC++: OpenCL and Level Zero; CUDA; HIP
  - hipSYCL / Open SYCL: CUDA, HIP, Level Zero; OpenCL (pre-release)
  - Leverage existing compilers and profiling and debugging tools
- Standard C++ with a custom library
  - No need for extra support in linters, IDEs, etc.
- Logically similar to OpenCL
  - (Almost) no need to deeply modify existing code
- Standardized interop with native libraries

SYCL enablement plan (late 2020)

- Step 1:
  - Target oneAPI DPC++ / Intel GPUs, but stay standard-compliant
  - Extend the portability layer (device initialization, data transfers, etc)
  - Remove code specific to CUDA/OpenCL
- Step 2:
  - Port kernels accounting for majority of run time
- Step 3:
  - Expand support to AMD GPUs
  - Port the rest of the kernels
  - Add support for GPU-aware MPI
  - Optimize kernels and runtime

SYCL enablement plan

- Step 1:
  - Target oneAPI DPC++ / Intel GPUs, but stay mostly standard-compliant
  - Extend the portability layer (device initialization, data transfers, etc)
  - Remove code specific to CUDA/OpenCL: still ongoing…
- Step 2:
  - Port kernels accounting for majority of run time
- Step 3:
  - Expand support to hipSYCL; AMD and NVIDIA GPUs
  - Port the rest of the kernels
  - Add support for GPU-aware MPI
  - Optimize kernels and runtime: we're here

Automatic conversion?

- SYCLomatic: automated CUDA⇨SYCL conversion tool
- ChatGPT: also works, at least for simple cases

- We want to have both CUDA and SYCL in the same codebase
- Already have abstraction layer for Device, Queue, etc
  - Supports CUDA and OpenCL
  - CUDA kernels heavily optimized for NVIDIA
  - Rewriting kernels is ~trivial
- Conclusion: manual porting

GPU framework comparison

	NVIDIA CUDA	OpenCL	SYCL
Scheduling	in-order queue or explicit DAG	in-order and out-of-order queues	implicit DAG and in-order queues
Synchronization event	separate barrier	associated with a task or a barrier	associated with a task
Timing measurement	regions	of a single event	of a single event
Timing enablement	at event creation	at queue creation	at queue creation
Device selection	stateful per-thread	explicit in each call	explicit in each call
Native float3 size	12 bytes	16 bytes	16 bytes

GPU framework comparison

We already had an abstraction layer

Scheduling	in-order queue or explicit DAG	in-order and out-of-order queues	implicit DAG and in-order queues
Synchronization event	separate barrier	associated with a task or a barrier	associated with a task
Timing measurement	regions	of a single event	of a single event
Timing enablement	at event creation	at queue creation	at queue creation
Device selection	stateful per-thread	explicit in each call	explicit in each call
Native float3 size	12 bytes	16 bytes	16 bytes

SYCL Buffers or USM

- Buffers prevent bugs and improve performance
  - At least in theory, the runtime must be smart
- GROMACS is built around in-order queues, with explicit synchronizations
  - Additional divergence between backends when using buffers

- Solution: use in-order queues and USM instead
- Bonus:
  - Accessors are hard to optimize for compiler
  - Easier interop with native libraries

Synchronization Events

- We use barriers in CUDA and OpenCL, but SYCL does not have them
- Event can be recorded far from the last submission
  - Not easy to tell which operation should be used for synchronization

- Solution: Use extensions to mark events:
  - oneAPI: `SYCL_EXT_ONEAPI_ENQUEUE_BARRIER`
  - hipSYCL: `hipSYCL_enqueue_custom_operation` to submit empty jobs acting as barriers

Synchronization Events

- Using vendor extensions to mark events
  - Same logic for all backends!
  - Code no longer fully standard-compliant
  - Works well with some extensions (hipSYCL's coarse-grained events)
  - Not compatible with some other extensions (oneAPI's SYCL Graph)
  - Performance issues with Level Zero

- Possibly will have to add a standard-compliant solution based on manually tracking the last recorded event

Other differences to keep in mind

- Exceptions vs return codes
- Different and variable (for Intel) sub-group sizes
- Thread indexing order:
  - CUDA and OpenCL: thread (x, y, z) is adjacent to (x+1, y, z)
  - SYCL: thread (x, y, z) is adjacent to (x, y, z+1)
- No SYCL implementation is fully standard-compliant yet
  - It's getting better

Interoperability in practice: FFT

- With USM, we have native device pointers
- Can mix-and-match native and SYCL kernels / API

- Intel GPUs: oneMKL or Double-Batched FFT
- AMD GPUs: rocFFT or vkFFT via `HIPSYCL_EXT_ENQUEUE_CUSTOM_OPERATION`
- NVIDIA GPUs: vkFFT via `HIPSYCL_EXT_ENQUEUE_CUSTOM_OPERATION`
- HeFFTe with oneMKL/rocFFT for multi-GPU decomposition

Performance portability in practice

- GROMACS uses SYCL to run on:
  - Intel GPUs via oneAPI,
  - AMD and NVIDIA GPUs via oneAPI and hipSYCL.
- Vendor-specific code:
  - Sub-group-size-dependent algorithms
  - FFT invocation, a lot of related CMake scripting
  - Extensions for synchronization events and runtime hints

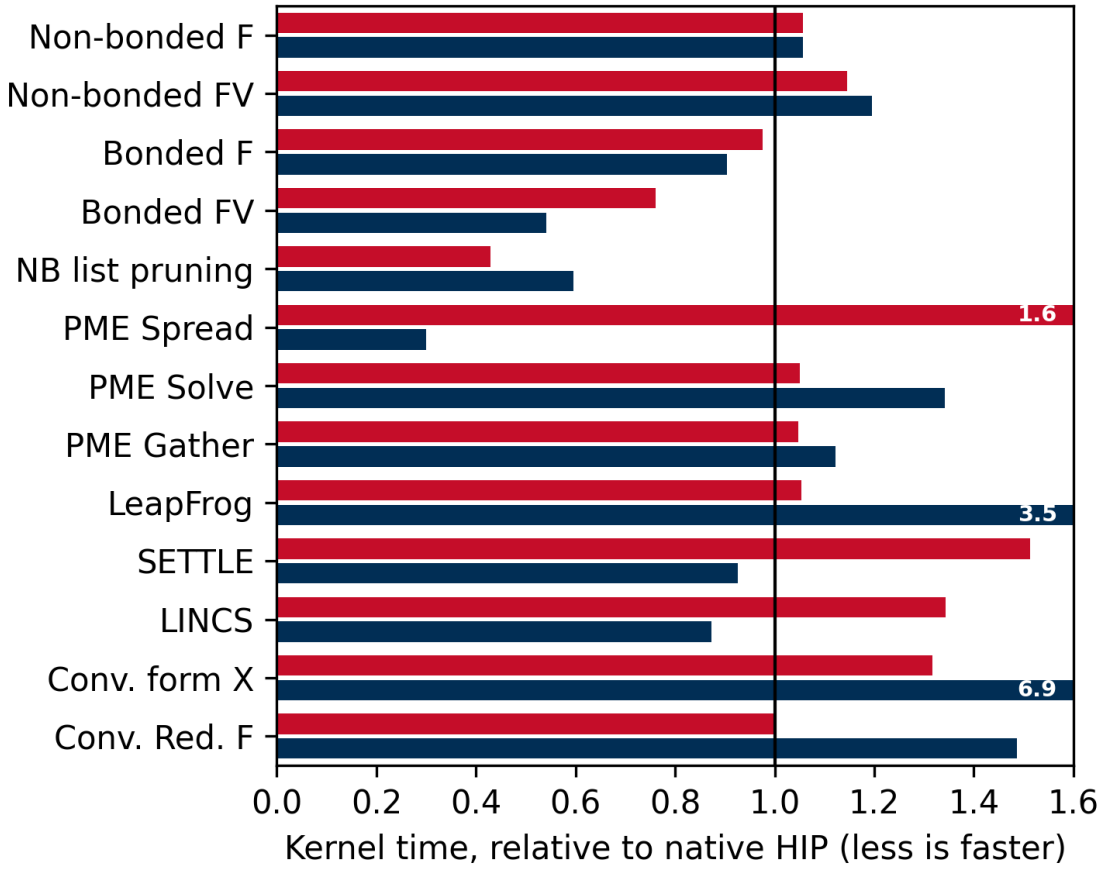
  - Overload some functions to use architecture-specific intrinsics
  - A data layout hack for two conflicting compiler issues

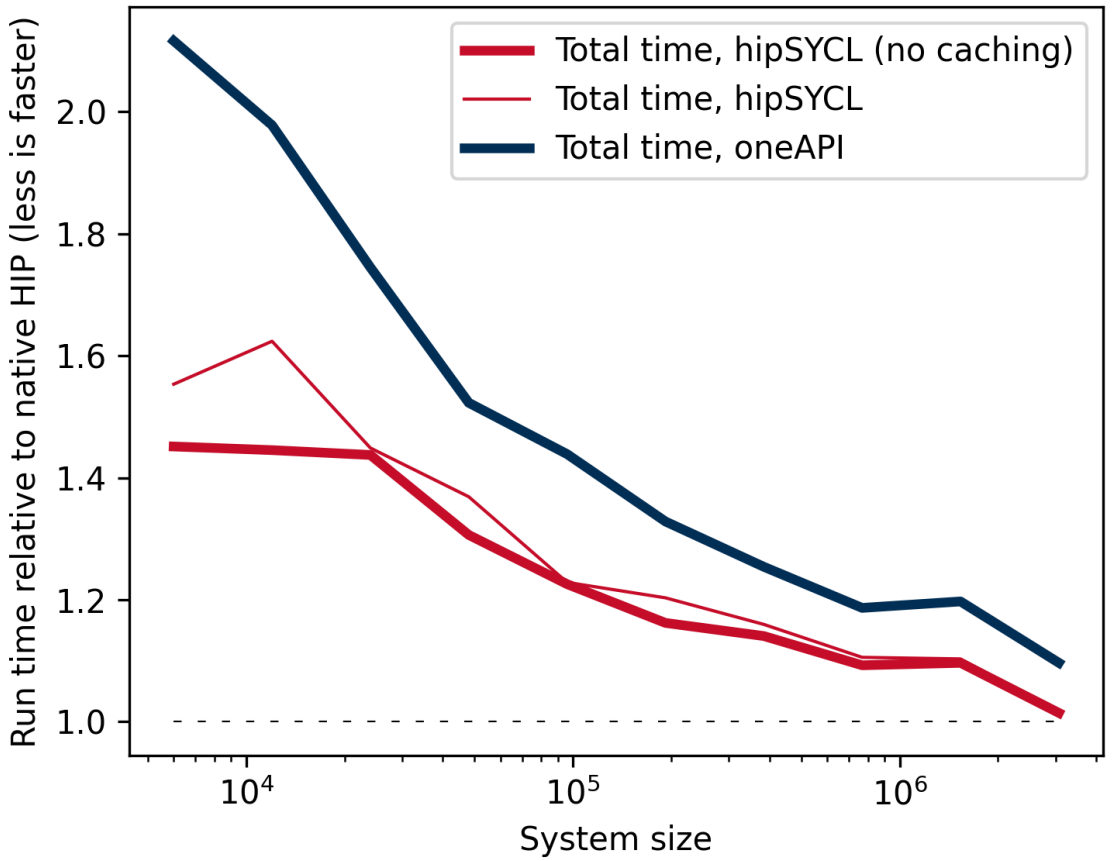
Performance portability in practice

- Kernel performance, compared to native CUDA/HIP/OpenCL:
  - Accessors waste registers, use raw pointers
  - Complex kernels might require some profiling
    - Typically, missed optimizations
    - Not too hard to get within 15%
  - Less complex kernels
- Extra runtime overhead when dealing with small kernels
  - Not an issue for larger kernels

Performance portability in practice

MI250X, ROCm 5.3.3, hipSYCL and oneAPI vs HIP native

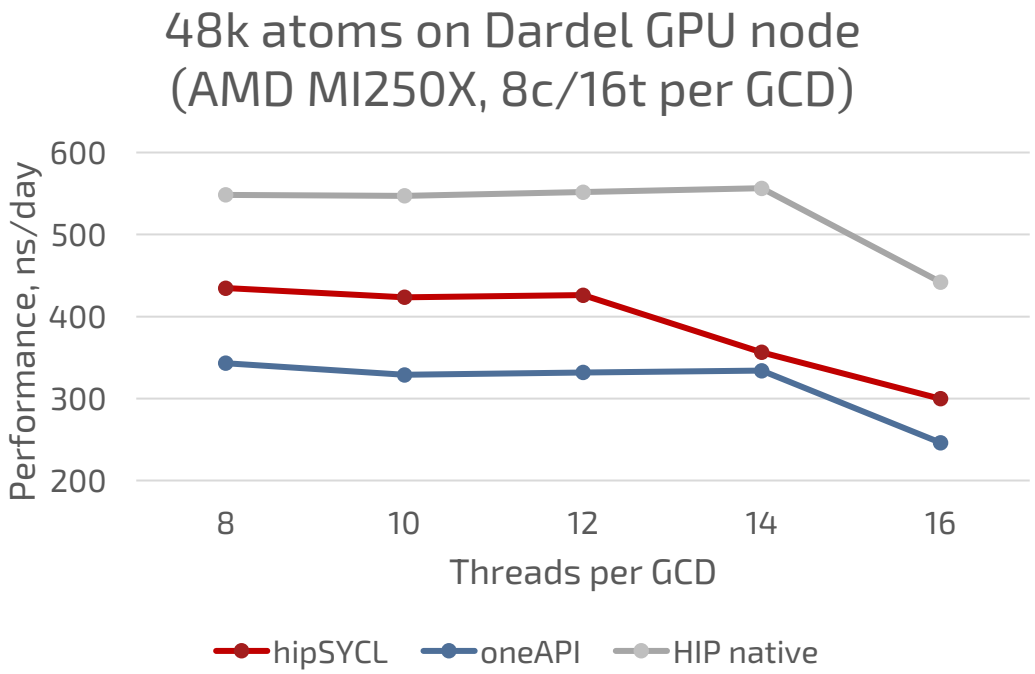




Performance portability in practice

- When dealing with small kernels (<10 µs), beware of runtime overheads

- Solutions:
  - hipSYCL's coarse-grained events
  - hipSYCL's instant submission mode
  - hipSYCL's caching control
  - oneAPI's discard_event queues
  - oneAPI's SYCL graphs
  - Poke runtime developers ☺



Results

- SYCL is a feature-complete GPU backend in GROMACS

- Support GPUs from all three vendors with minimal specialization
  - Performance typically within 20% of **highly-optimized** native code
  - Downside: any change of code requires extensive testing

- Production ready:
  - Running on LUMI today, efficiently scaling across multiple nodes

Conclusions

- SYCL allow writing performant, portable, maintainable code
- … but running fast is never easy
  - Still need vendor-specific code branches to get the best performance
  - Runtime might behave sub-optimally by default
- Still, in many cases, it just works!
- New versions can bring both improvements and regressions

Acknowledgements

- Intel Corporation
- Mark Abraham , Heinrich Bockhorst and Roland Schulz (Intel)
- Aksel Alpay (Heidelberg University Computing Centre)
- GROMACS dev team, in particular Szilárd Páll

Learn more

- https://www.gromacs.org/
- https://gromacs.bioexcel.eu/
- https://manual.gromacs.org/

- Páll *et al.*, J. Chem. Phys. 153, 134110 (2020)

- **If you have questions: ask them on Slack**
  - **or email me: andreyal@kth.se**