FAST. FLEXIBLE. FREE.
GROMACS

oneAPI Case Study: GROMACS

Andrey Alekseenko

KTH Royal Institute of Technology & SciLifeLab
Stockholm, Sweden

GROMACS

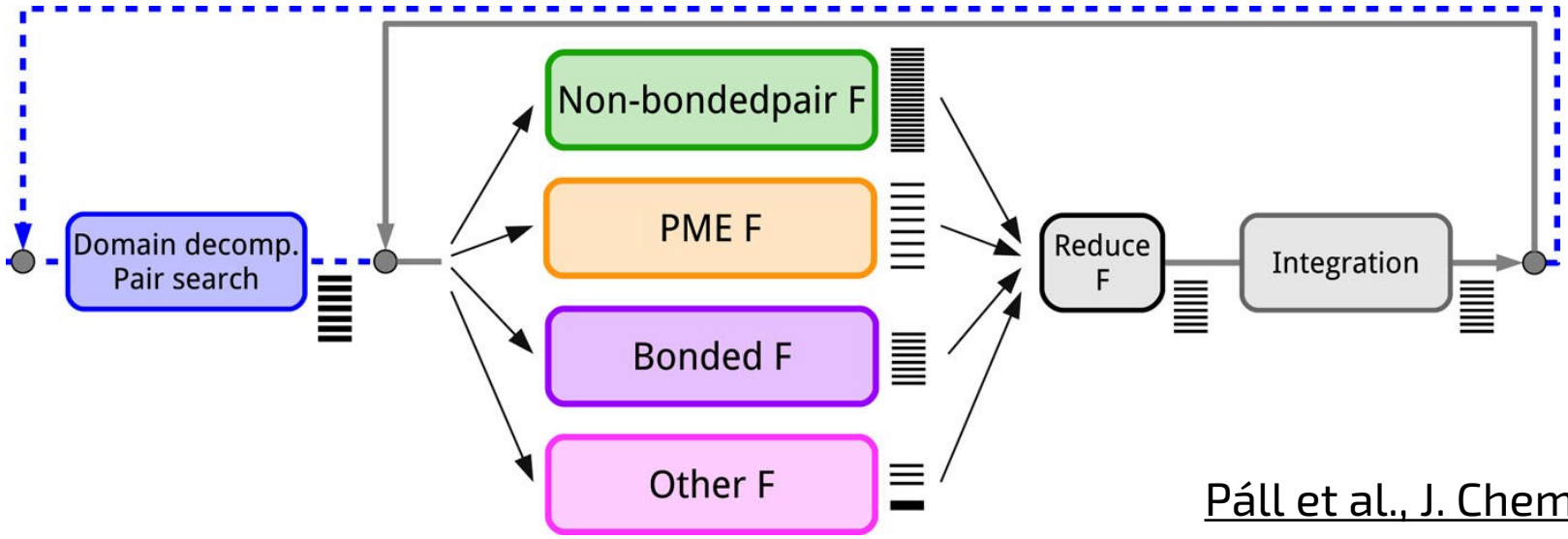
- Open source molecular dynamics engine
- One of the most used HPC codes worldwide
- High-performance for a wide range of modeled systems
- … and on a wide range of platforms:
  - from supercomputers to laptops (Folding@Home)
  - x86, x86-64, ARM, POWER, SPARC, RISC-V
  - 11 SIMD backends
  - NVIDIA,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

GROMACS 2023 (upcoming)

- (Mostly) modern C++17 codebase
  - 449k 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



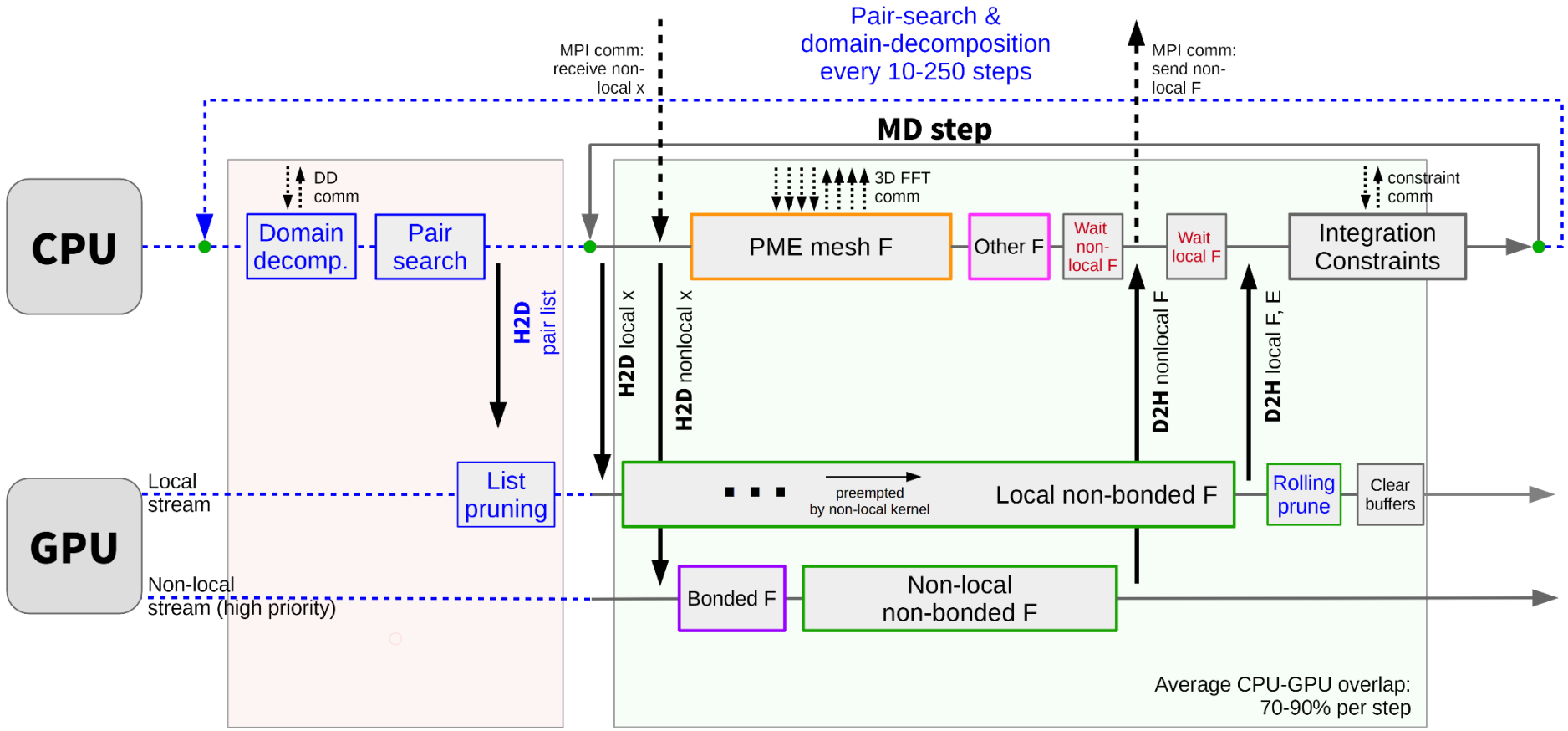
Páll et al., J. Chem. Phys. 153, 134110 (2020)

Heterogeneous parallelization

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

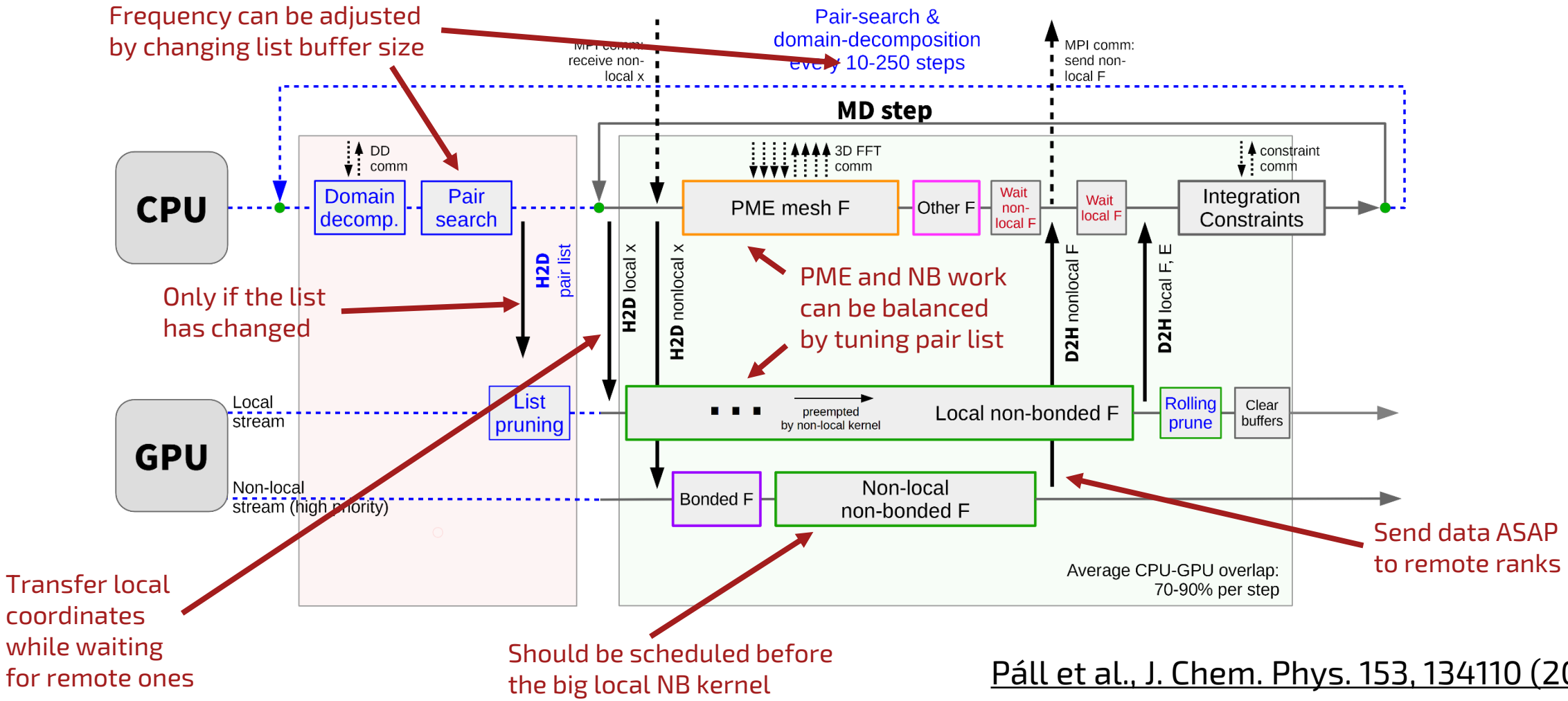
- Optimal offloading scheme depends on simulated system
  - And on available hardware
- Must be maintainable

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)

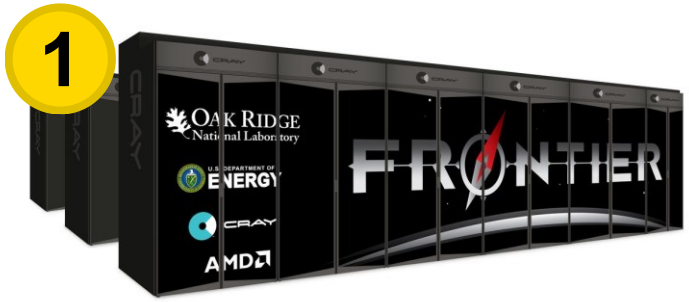
GPU feature support in GROMACS 2020

	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

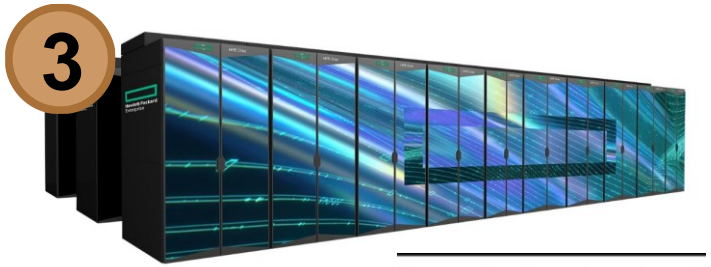
Why another GPU framework?

AMD Instinct GPU

Intel Ponte Vecchio GPU



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

Why SYCL?

- Open standard, free (libre) implementations
- Implemented on top of existing backend
  - Intel® oneAPI DPC++: OpenCL and LevelZero; CUDA; HIP
  - hipSYCL: CUDA, HIP; LevelZero (via DPC++)
  - Leverage existing profiling and debugging tools
  - And device compilers
- 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

SYCL enablement plan (late 2020)

- Step 1:
  - Target oneAPI DPC++ / Intel GPUs, but stay standard-compliant
  - Device detection and initialization
  - 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
  - Device detection and initialization
  - 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?

- 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
  - OpenCL kernels have Intel-optimized code paths
  - 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 pseudo-task	associated with a task or a pseudo-task	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 pseudo-task	associated with a task or a pseudo-task	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

	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 pseudo-task	associated with a task or a pseudo-task	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

DAG-based scheduling

- Good: Prevent bugs and improve performance
- Bad: GROMACS is built around for in-order queues, with explicit barrier synchronizations:
  - Performance: synchronizing twice
  - Correctness: device-to-host copies
- Bad: Runtime must be smart
- Ugly: Additional divergence between backends

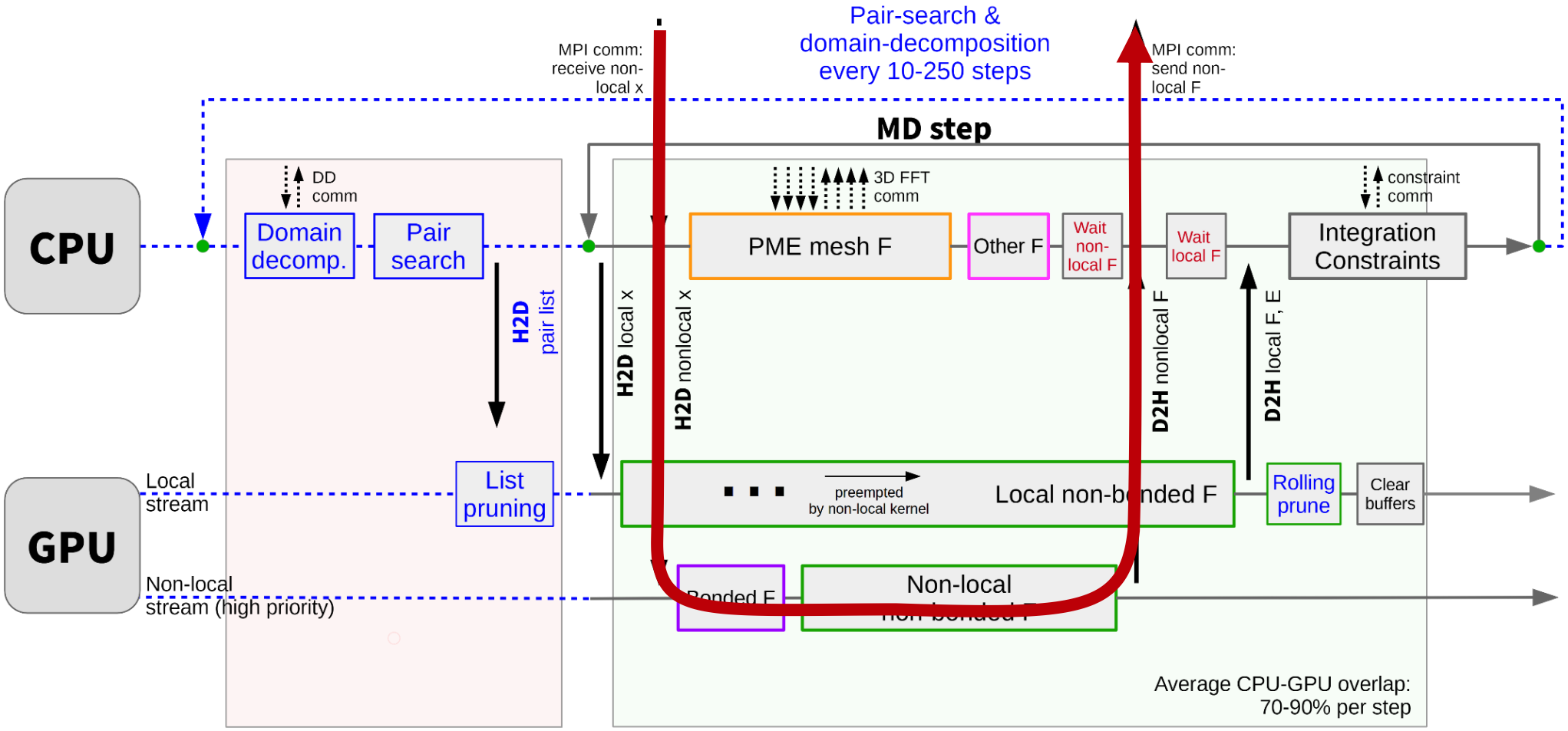
DAG-based scheduling

- Good: Prevent bugs and improve performance
- Bad: GROMACS is built around for in-order queues, with explicit barrier synchronizations:
  - Performance: synchronizing twice
  - Correctness: device-to-host copies
- Bad: Runtime must be smart
- Ugly: Additional divergence between backends

DAG-based scheduling

- Good: Prevent bugs and improve performance
- Bad: GROMACS is built around for in-order queues, with explicit barrier synchronizations:
  - Performance: synchronizing twice
  - Correctness: device-to-host copies
- Bad: Runtime must be smart
- Ugly: Additional divergence between backends

DAG-based scheduling



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

DAG-based scheduling

- Good: Prevent bugs and improve performance
- Bad: GROMACS is built around for in-order queues, with explicit barrier synchronizations:
  - Performance: synchronizing twice
  - Correctness: device-to-host copies
- Bad: Runtime must be smart
- Ugly: Additional divergence between backends

DAG-based scheduling

- No, just use in-order queues and USM

- Bonus:
  - Accessors hard to optimize for compiler
  - Easier interop with GPU-aware MPI

GPU framework comparison

			
Scheduling	in-order queue or explicit DAG	in-order and out-of-order queues	implicit DAG and in-order queues
Synchronization event	separate pseudo-task	associated with a task or a pseudo-task	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	by special function	explicit in each call	explicit in each call
Native float3 size	12 bytes	16 bytes	16 bytes

Synchronization Events

- Event can be recorded far from the last submission
  - Not easy to tell which operation should be used for synchronization

- Custom extensions to mark events:
  - oneAPI DPC++: `SYCL_EXT_ONEAPI_ENQUEUE_BARRIER`
  - hipSYCL: `hipSYCL_enqueue_custom_operation` to submit empty jobs acting as barriers

- hipSYCL's coarse-grained events

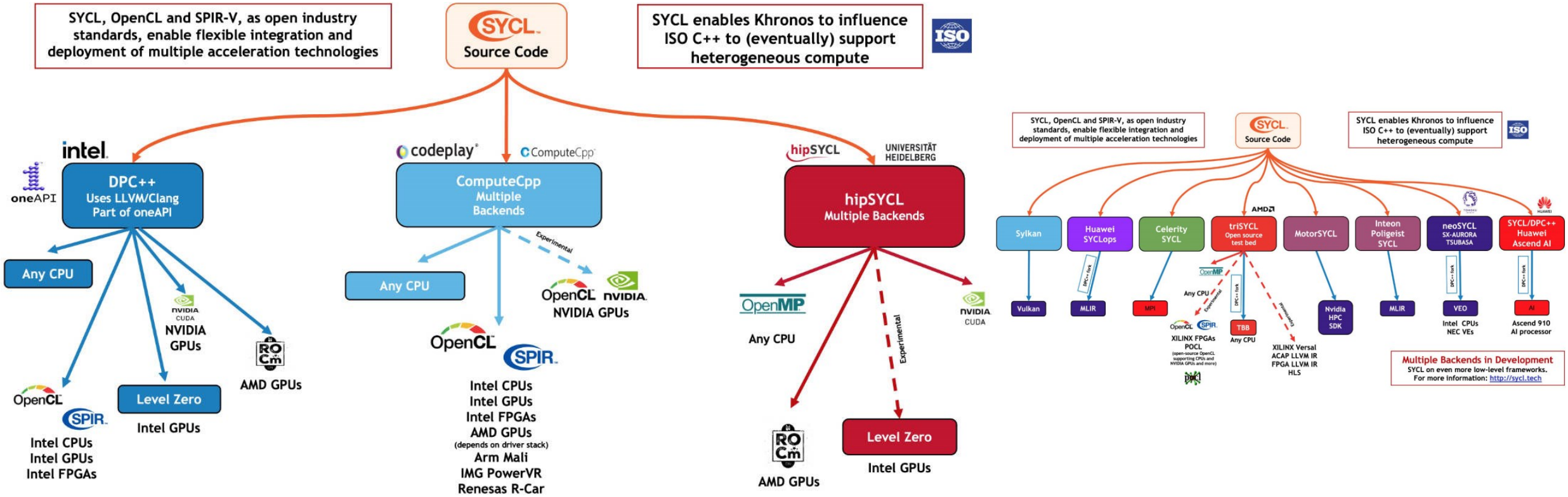
GPU framework comparison

			
Scheduling	in-order queue or explicit DAG	in-order and out-of-order queues	implicit DAG and in-order queues
Synchronization event	separate pseudo-task	associated with a task or a pseudo-task	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

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
  - Some standardized features still implemented as extensions
  - It's getting better
- No SYCL implementation is fully optimized
  - Less of an issue for compilers, more of an issue for the runtime

SYCL beyond oneAPI



https://www.khronos.org/sycl/

Portability in practice: FFT

- 3D real-to-complex, forward and backward FFT

- Intel GPUs: oneMKL
- AMD GPUs: rocFFT/vkFFT via `HIPSYCL_EXT_ENQUEUE_CUSTOM_OPERATION`
- NVIDIA GPUs: vkFFT via `HIPSYCL_EXT_ENQUEUE_CUSTOM_OPERATION`

- Future: HeFFTe to decompose FFT over multiple nodes?

Portability in practice: hipSYCL

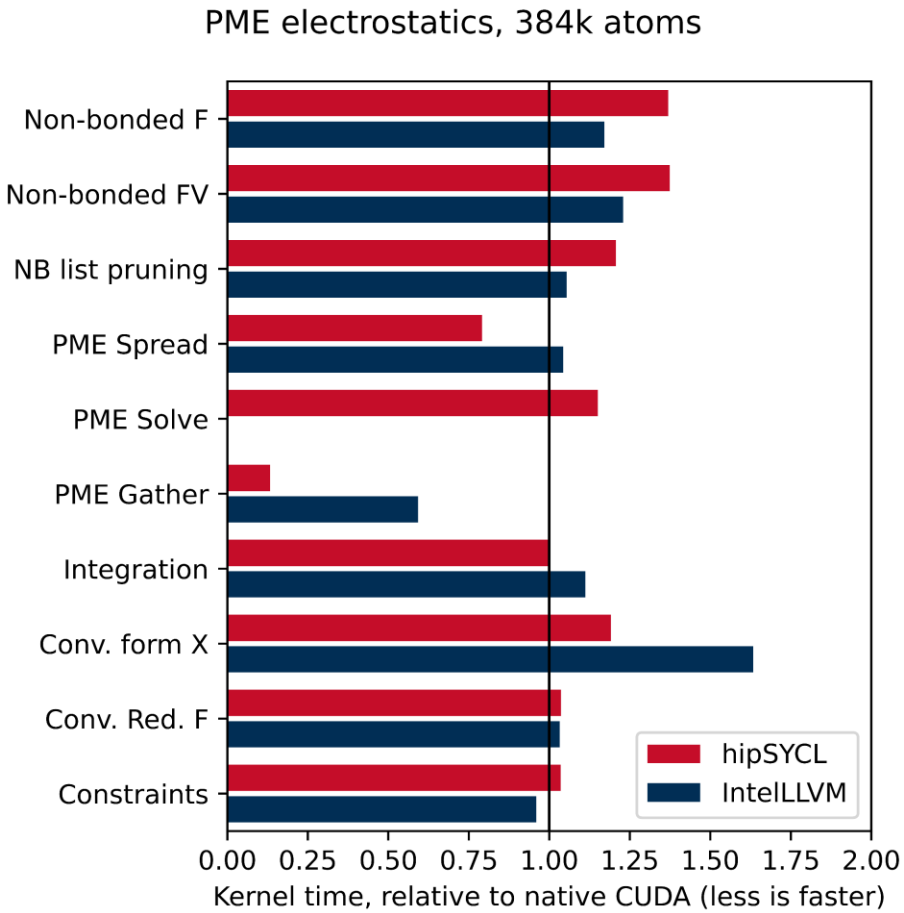
- At start, only Intel oneAPI DPC++ and Intel GPUs supported
  - hipSYCL added later to target AMD devices
- Effort:
  - Workarounds due to backend / compiler issues
    - Different parts of SYCL 2020 implemented
  - Fix a few bugs not triggered with oneAPI
  - CMake scripting
  - Kernel optimizations mostly ported from OpenCL
    - Still, some time with profiler was required
  - Runtime profiling/tuning
    - When the GPU FLOPS are not the bottleneck

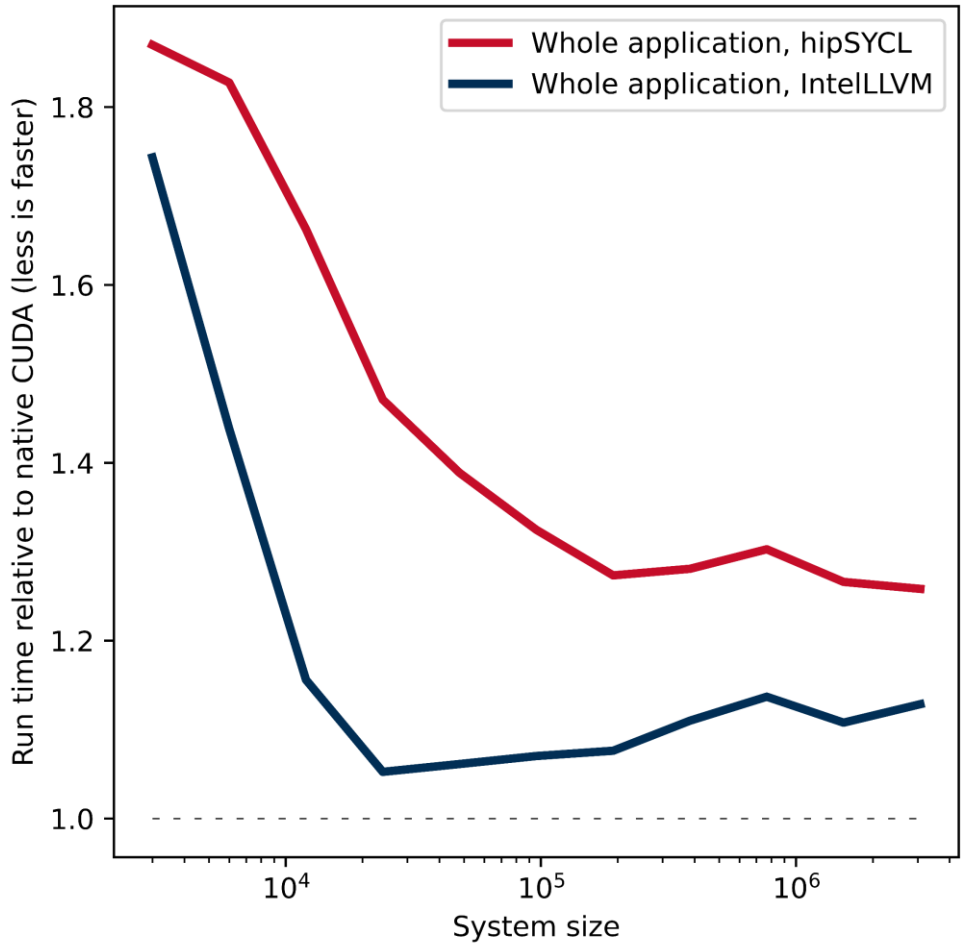
Portability in practice: results

- GROMACS can use SYCL to run on:
  - Intel GPUs via oneAPI,
  - AMD and NVIDIA GPUs via oneAPI and hipSYCL
- Performance, compared to native CUDA/HIP/OpenCL:
  - Complex kernels are slower
  - Less complex kernels on par, sometimes faster
  - Extra runtime overhead
- Relatively little vendor-specific code
  - Sub-group-size-dependent algorithms
  - Workarounds for compiler issues
  - FFT invocation

Performance: NVIDIA

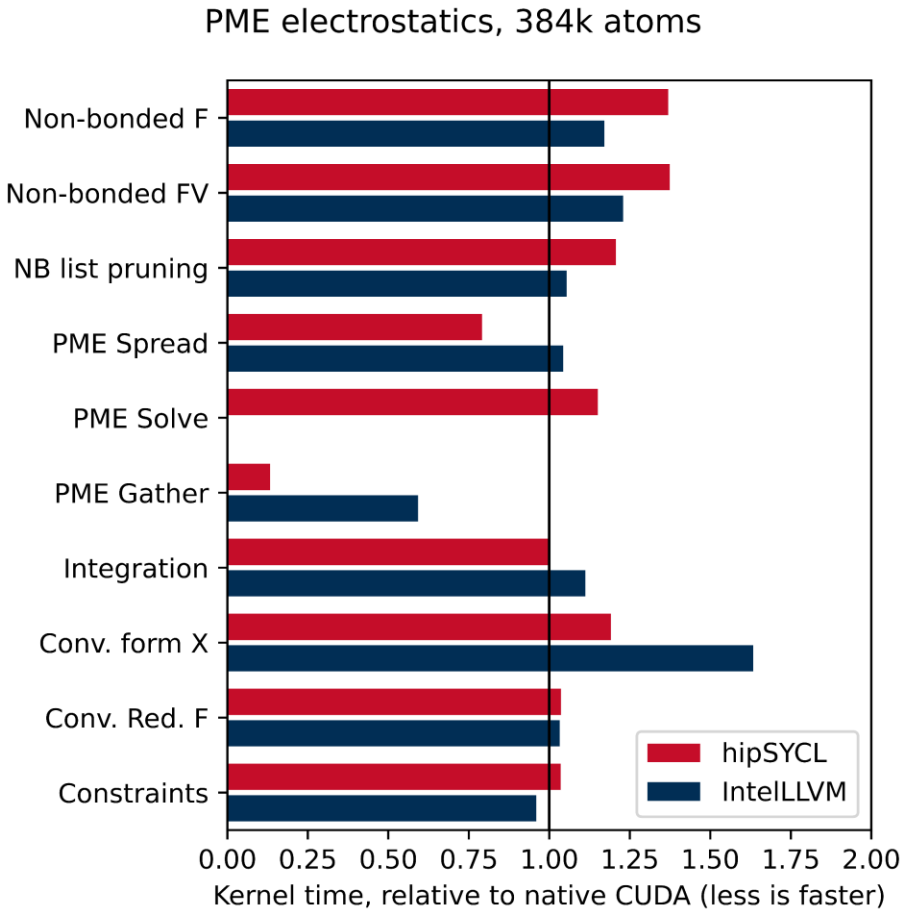
V100, CUDA 11.5, hipSYCL develop and IntelLLVM vs CUDA

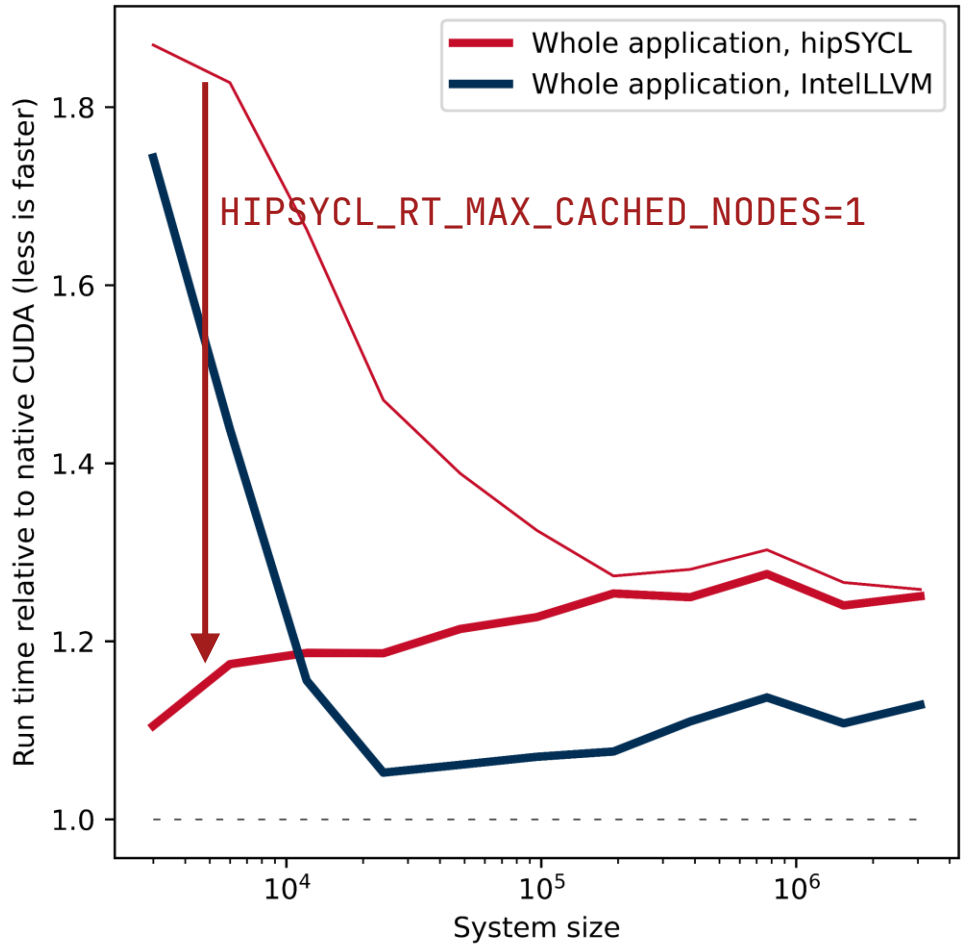




Performance: NVIDIA

V100, CUDA 11.5, hipSYCL develop and IntelLLVM vs CUDA

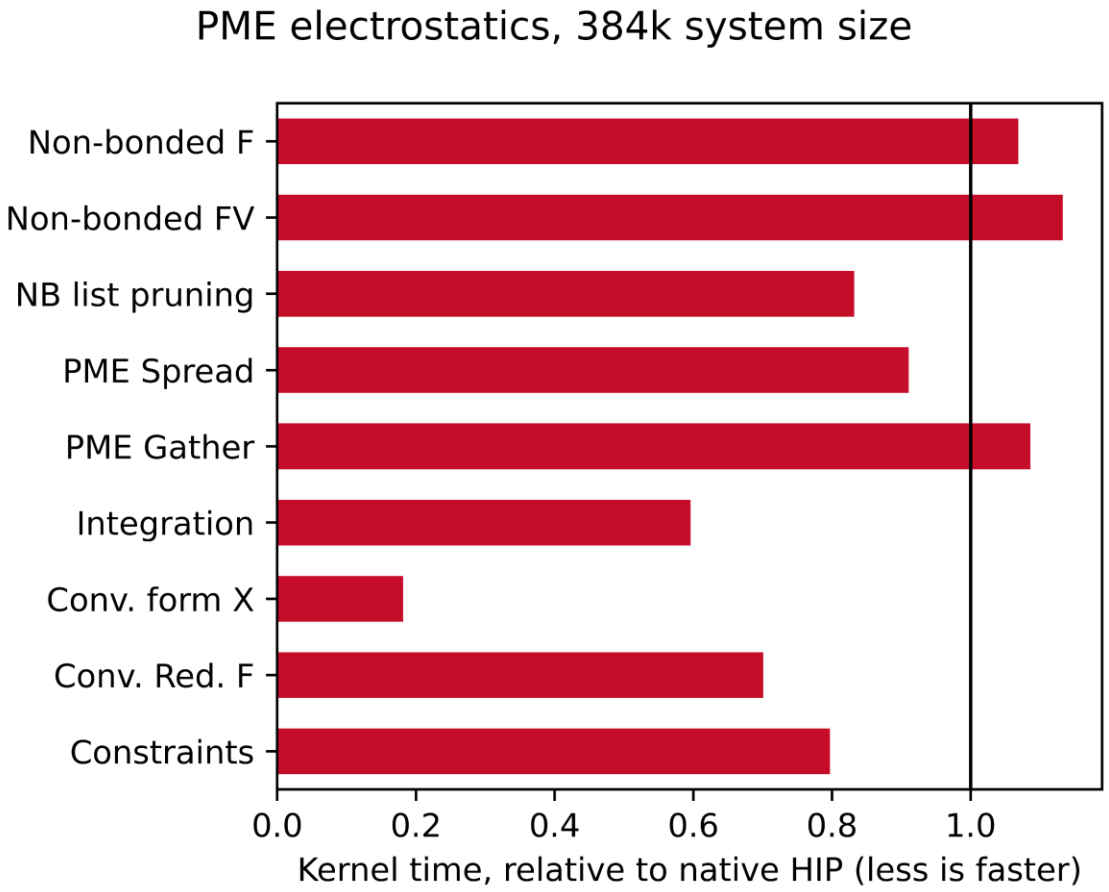


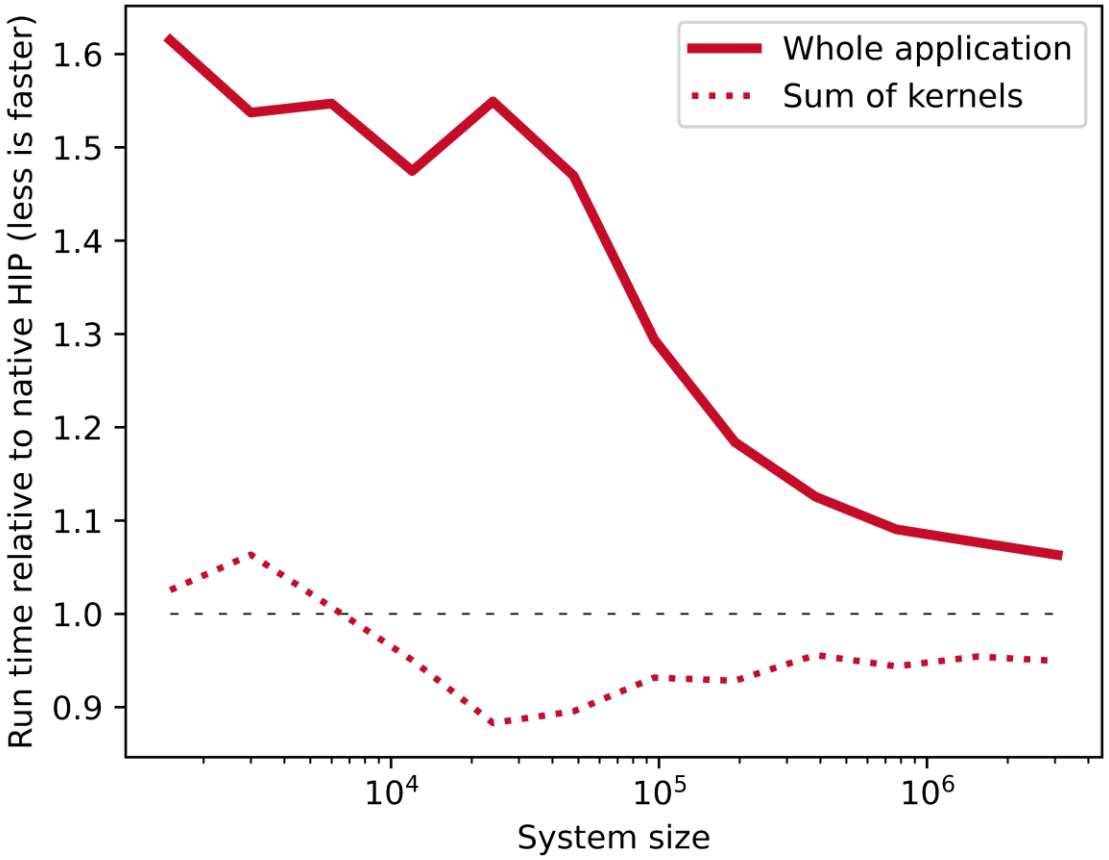


Performance: AMD

older GROMACS version

MI100, ROCm 4.5.2, hipSYCL 0.9.2 vs experimental HIP port by AMD/StreamHPC

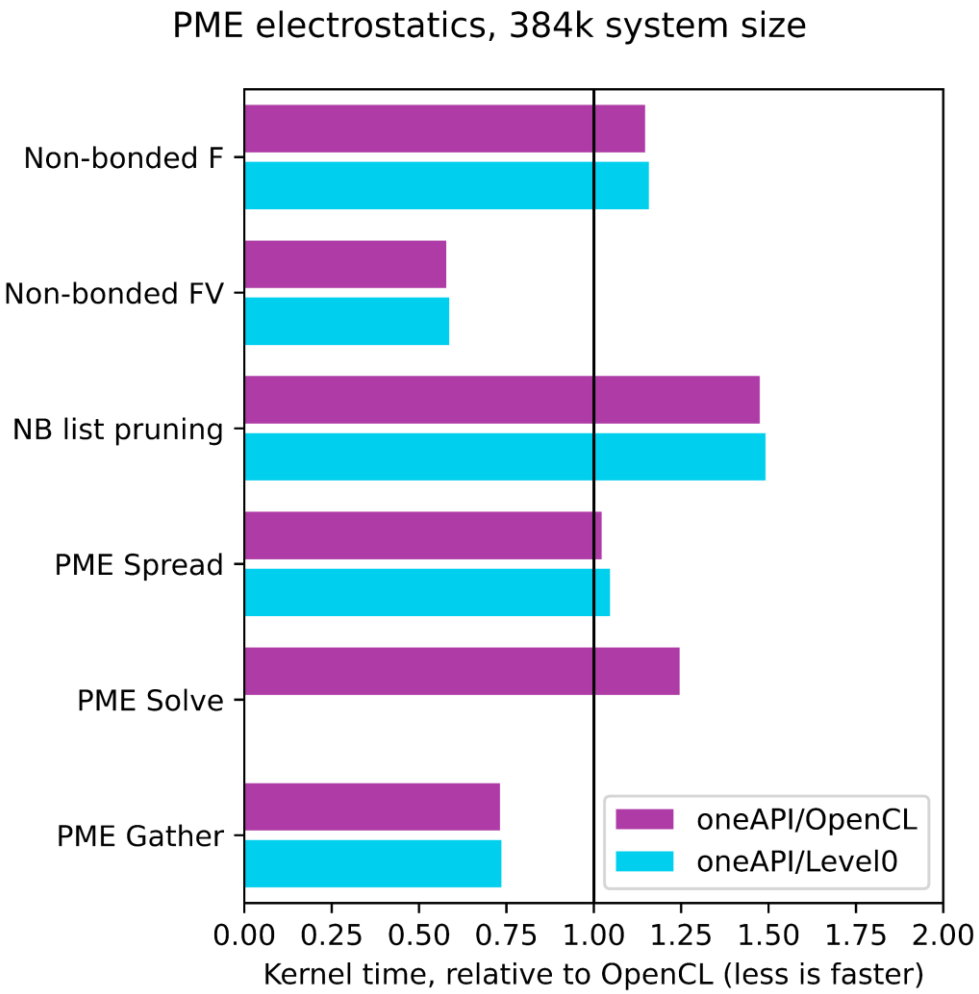


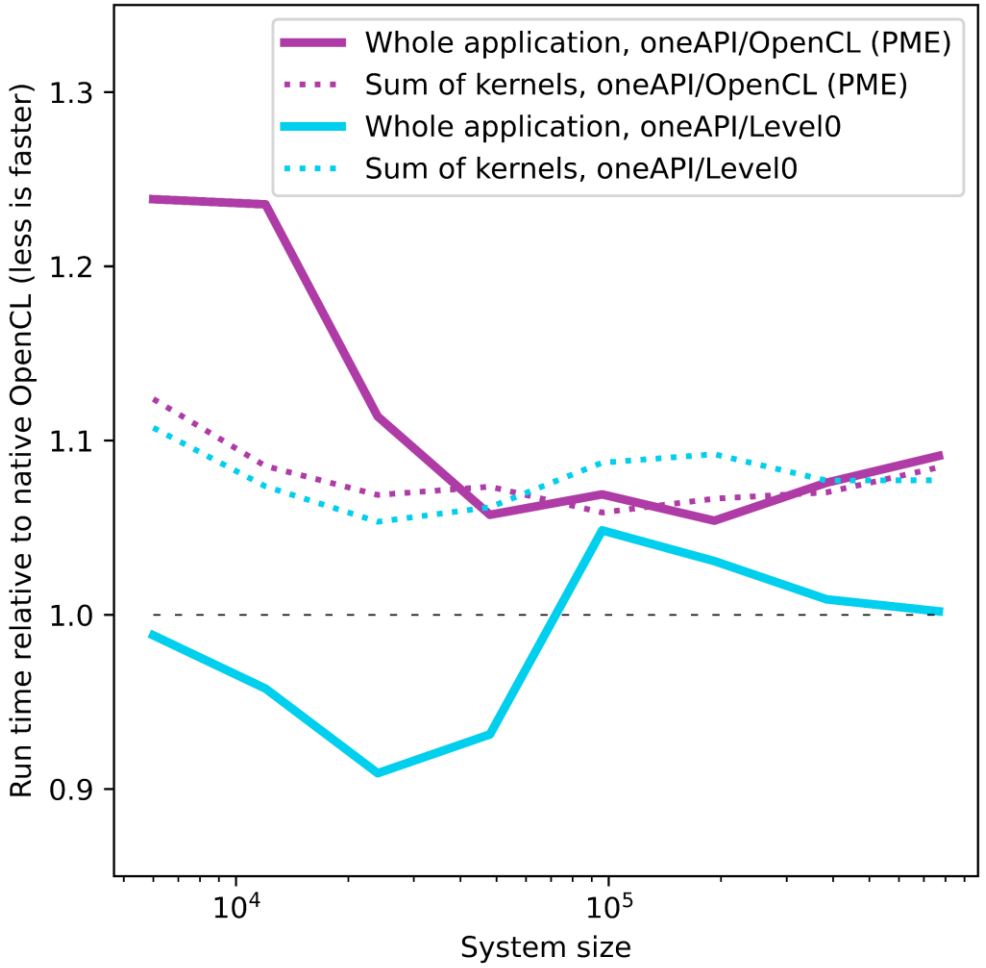


Performance: Intel

older GROMACS version, low-end GPU

Xe MAX (DG1), oneAPI 2022.1, oneAPI (LevelZero, OpenCL) vs OpenCL





Missing features / Wishlist

- Queue/task priorities
- Fine control of host scheduling
- Hardware topology information
- Inter-node communication: Celerity?
- Libraries: oneMKL?
- Robustness: installation, error messages, …

GPU feature support in GROMACS 2023

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

Conclusions

- “Write once, run anywhere” mostly works
  - Trivial changes to support all three major vendors with oneAPI and hipSYCL
- But running fast is not easy
  - Still need vendor-specific code branches to get high performance
  - Runtime might behave sub-optimally by default
- API is similar to OpenCL in spirit, but usually nicer
- The whole ecosystem is rapidly evolving

Acknowledgements

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

Learn more

- https://gromacs.org/
- https://www.gromacs.org/Support/GMX-Developers_List
- https://gitlab.com/gromacs/gromacs/
- https://manual.gromacs.org/documentation/2022-rc1/index.html

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

- **If you have questions: andrey.alekseenko@scilifelab.se**