



# A oneAPI Case Study: GROMACS



Andrey Alekseenko

KTH Royal Institute of Technology & SciLifeLab  
Stockholm, Sweden

# GROMACS

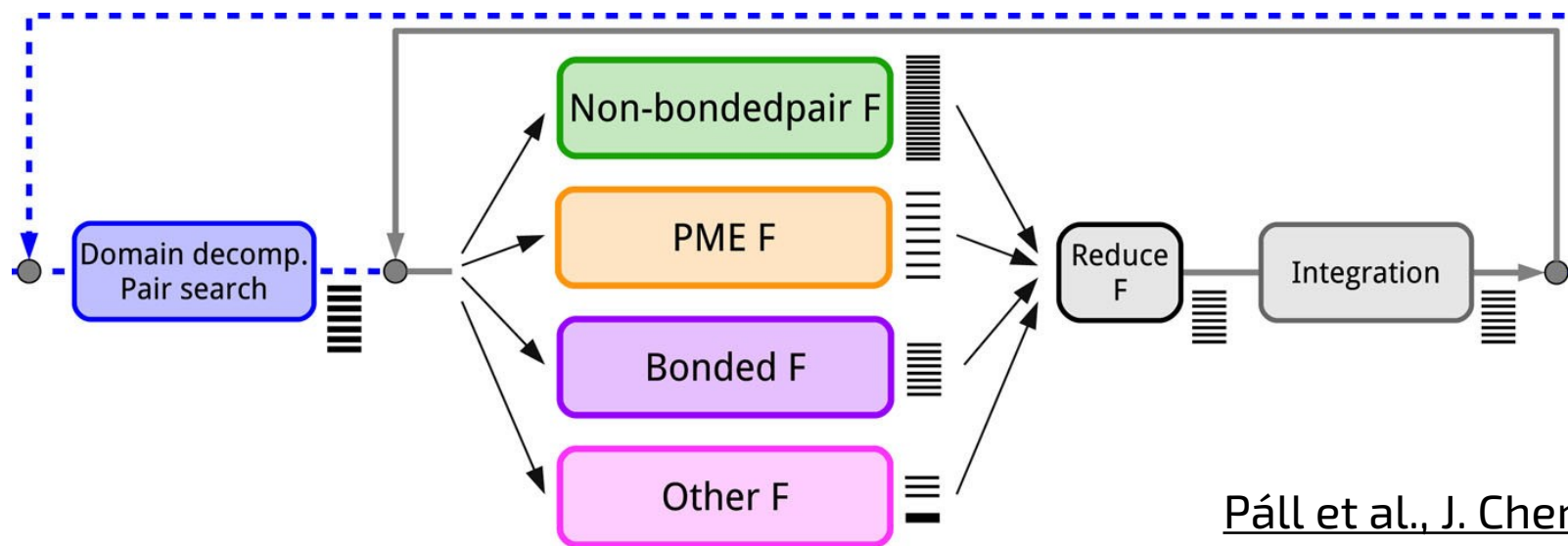
- Open source molecular dynamics engine
- One of the most used HPC codes worldwide
- High-performance for a wide range of modeled systems
- ... and on a wide range of platforms:
  - from supercomputers to laptops (Folding@Home)
  - x86, x86-64, ARM, POWER, SPARC
  - 11 SIMD backends
  - NVIDIA, AMD,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

# GROMACS 2022

- (Mostly) modern C++17 codebase
  - 439k 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: NVIDIA, AMD, Intel
  - SYCL: NVIDIA, AMD, Intel

# Molecular dynamics

- Iterative problem
  - Like N-body, but with fancier physics
- One step 1 fs, need to simulate  $\mu\text{s}$  to ms
  - $10^9$ - $10^{12}$  steps

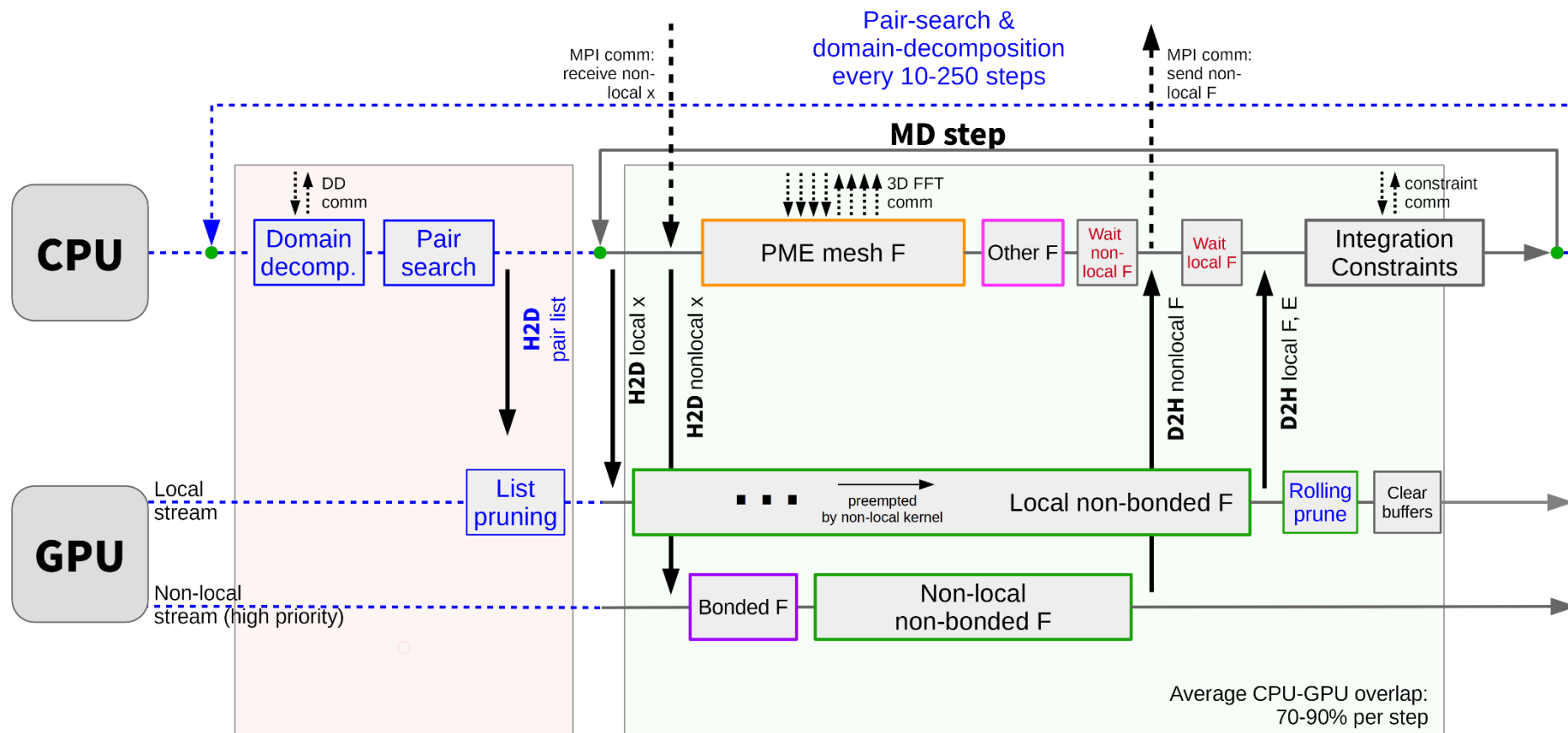


Páll et al., J. Chem. Phys. 153, 134110 (2020)

# Heterogeneous parallelization

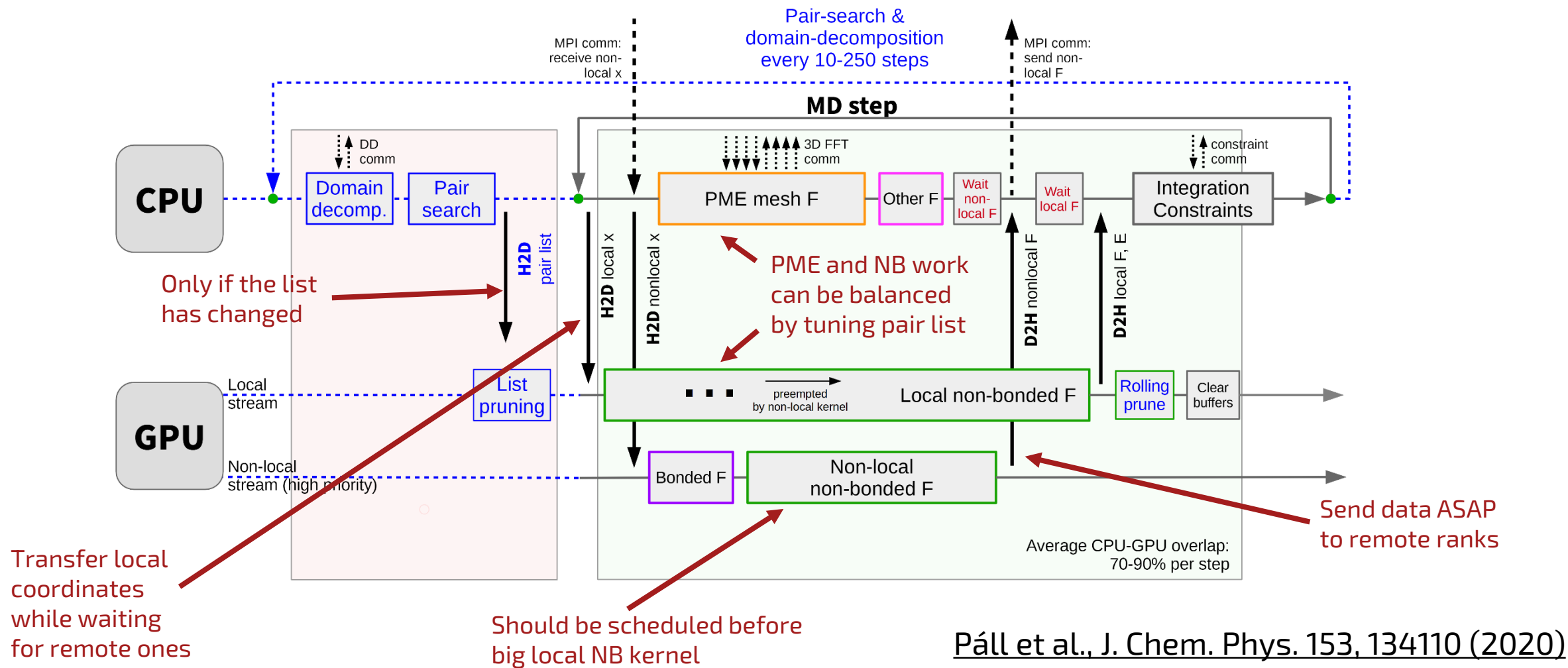
- Latency
- Minimizing CPU and GPU stalls
- Resources are not infinite
- Host-to-device data exchange is costly
  - Internode even costlier
- Optimal offloading scheme depends on simulated system
  - And on the hardware

# Molecular dynamics: real schedule



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

# Molecular dynamics: real schedule



# GPU feature support in GROMACS 2020

Released January 2020



Device detection	✓	✓
Non-bonded	✓	✓
PME	✓	✓
Update (integration)	✓	X
Bonded	✓	X
Direct GPU-GPU comm	✓*	X
Hardware support	NVIDIA	NVIDIA, AMD, Intel



# Why another GPU framework?

HPC systems coming in 202<sup>2</sup>

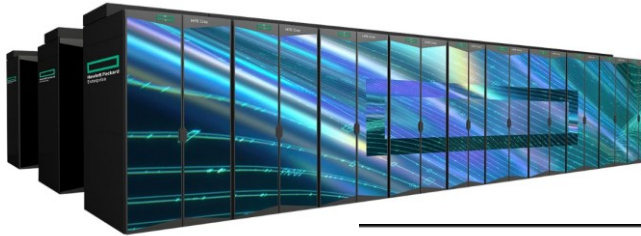
AMD Instinct GPU



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

# SYCL

- Open standard, free (libre) implementations
- Implemented on top of existing backend
  - Intel® oneAPI DPC++: OpenCL and LevelZero; CUDA
  - 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

# Comparison of GPU frameworks

	 <b>NVIDIA</b> <small>CUDA</small>	 <b>OpenCL</b> <sup>™</sup>	 <b>SYCL</b> <sup>™</sup>
Maturity level	✓	✓	X
Open standard	X	✓	✓
HW support: vendor	NVIDIA	NVIDIA, AMD, Intel	Intel
HW support: 3 <sup>rd</sup> party	-	-	AMD*, NVIDIA
Single-source model	✓	X	✓
Syntax	C++17/CUDA	C99/OpenCL	C++17
 In GROMACS	Main GPU backend for NVIDIA GPUs.	Primary support for AMD and Intel GPUs, some support for NVIDIA.	Early support in 2021. Full support in 2022 (experimental).

# SYCL enablement plan

- Use oneAPI DPC++, but try to stay standard-compliant
- Step 1:
  - Device detection and initialization
  - Remove (most) hacks specific for CUDA/OpenCL
- Step 2:
  - Port kernels accounting for 95% of runtime:
    - Non-bonded force & energy kernels
- Step 3:
  - Port update kernels to stay fully on the device for ~100 iterations
  - PME electrostatics (kernels and FFT)

# DPC++ Compatibility Tool

- We want to have both CUDA and SYCL in the same codebase
- Existing abstraction layer for Device, Queue, etc
  - Already supports CUDA and OpenCL
  - Work on expanding this abstraction layer!
- CUDA kernels heavily optimized for NVIDIA
  - OpenCL kernels have Intel-optimized code paths
  - Rewriting kernels is ~trivial
- Conclusion: manual porting

# SYCL version requirements

- Kernels already highly optimized:
  - Subgroup-level functionality
  - Floating-point atomics
- SYCL 1.2.1 is not enough!
- SYCL 2020 published in February 2021
  - Porting work started in September 2020
- No SYCL implementation is fully standard-compliant
  - Had to rely on oneAPI extensions
    - Most became part of SYCL2020

# GPU framework comparison



	in-order queue or explicit DAG	in-order and out-of-order queues	implicit DAG and in-order queues*
Scheduling			
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
Resource management	manual	manual	RAII



# GPU framework comparison

We already had an abstraction layer



	in-order queue or explicit DAG	in-order and out-of-order queues	implicit DAG and in-order queues*
Scheduling			
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
Resource management	manual	manual	RAII

# GPU framework comparison



	in-order queue or explicit DAG	in-order and out-of-order queues	implicit DAG and in-order queues <sup>*</sup>
Scheduling			
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
Resource management	manual	manual	RAII

# 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

## In-order queues:

- D2H Copy A
- D2H Copy B
- Enqueue event
- ...
- Wait for the event
- // both A and B completed

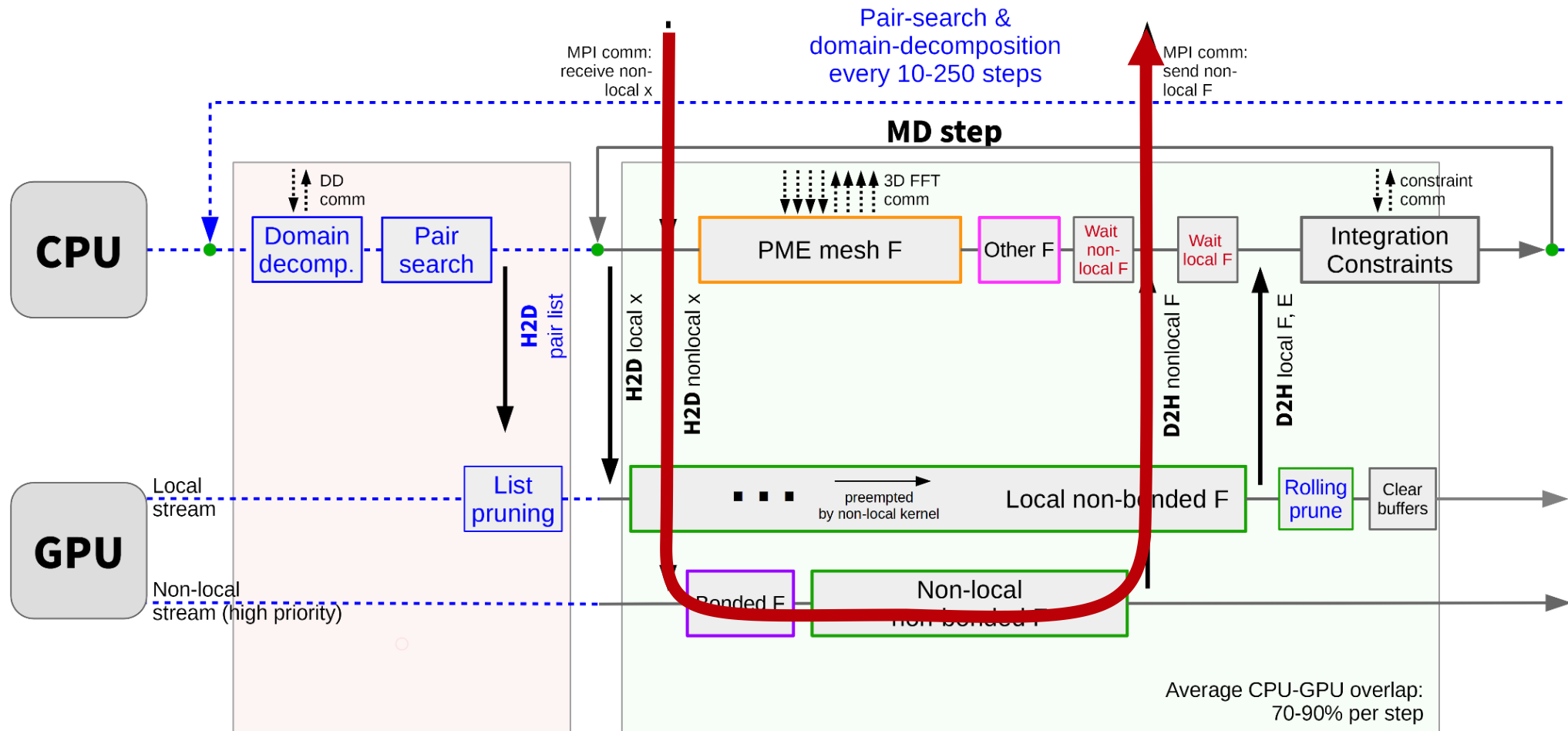
## DAG:

- D2H Copy A
- D2H Copy B
- Use the event from B
- ...
- Wait for the event
- // only B completed

# 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

- In 2020, USM was not yet standardized
  - Use buffers, treat them like a simple device memory
  - Explicitly synchronize HtoD and DtoH data copies
- Implicit DtoD synchronizations?
  - No performance benefits cf. explicit DtoD synchronizations
- In 2021, switched to USM and in-order queues
  - Accessors too demanding of compiler
  - Still keep buffers as an option

# GPU framework comparison



	in-order queue or explicit DAG	in-order and out-of-order queues	implicit DAG and in-order queues*
Scheduling			
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
Resource management	manual	manual	RAII

# Synchronization Events

- Inspired by CUDA, we create events once and reuse them
  - Event can be added to a list of dependencies, and enqueued later
- Event can be recorded far from the last submission
- Use an object to hold a reference to an event
- Custom extensions to mark events:
  - oneAPI DPC++: SYCL\_EXT\_ONEAPI\_ENQUEUE\_BARRIER
  - hipSYCL: HIPSYCL\_EXT\_QUEUE\_WAIT\_LIST

# GPU framework comparison



	in-order queue or explicit DAG	in-order and out-of-order queues	implicit DAG and in-order queues*
Scheduling			
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
Resource management	manual	manual	RAII

# Other differences

- Exceptions vs return codes
- Variable sub-group (warp) sizes
  - SIMD32 supported, but rarely optimal
- Launching kernel:
  - CUDA: (# work groups total; # items in work group)
  - OpenCL and SYCL: (# items total, # items in work group)
    - Or even (# items total)
- 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  $(x, y, z+1)$

# SYCL vs OpenCL

- No more duplicating structure definitions
- No more duplicating helper functions
- Templates instead of preprocessor:



```
#ifdef LJ_FORCE_SWITCH
#   ifdef CALC_ENERGIES
        calculate_force_switch_F_E(nbparam, c6, c12, inv_r, r2, &F_invr, &E_lj_p);
#   else
        calculate_force_switch_F(nbparam, c6, c12, inv_r, r2, &F_invr);
#   endif /* CALC_ENERGIES */
#endif /* LJ_FORCE_SWITCH */
```



```
if constexpr (props.vdwFSwitch)
{
    ljForceSwitch<doCalcEnergies>(
        nbparam, c6, c12, rInv, r2, &fInvR, &energyLJPair);
}
```

# SYCL vs OpenCL



```
#   ifndef LJ_COMB
__local int* atib = (__local int*)(LOCAL_OFFSET); //NOLINT(google-readability-casting)
#       undef LOCAL_OFFSET
#       define LOCAL_OFFSET (atib + c_nbnxnGpuNumClusterPerSupercluster * CL_SIZE)
#   else
__local float2* ljcpib      = (__local float2*)(LOCAL_OFFSET);
#       undef LOCAL_OFFSET
#       define LOCAL_OFFSET (ljcpib + c_nbnxnGpuNumClusterPerSupercluster * CL_SIZE)
#   endif
```

```
auto sm_atomTypeI = [&]() {
    if constexpr (!props.vdwComb)
    {
        return sycl::local_accessor<int, 2>(
            sycl::range<2>(c_nbnxnGpuNumClusterPerSupercluster, c_clSize), cgh);
    }
    else { return nullptr; }
}();
```



```
auto sm_ljCombI = [&]() {
    if constexpr (props.vdwComb)
    {
        return sycl::local_accessor<Float2, 2>(
            sycl::range<2>(c_nbnxnGpuNumClusterPerSupercluster, c_clSize), cgh);
    }
    else { return nullptr; }
}();
```

# SYCL vs OpenCL



```
#   ifndef LJ_COMB
__local int* atib = (__local int*)(LOCAL_OFFSET); //NOLINT(google-readability-casting)
#       undef LOCAL_OFFSET
#       define LOCAL_OFFSET (atib + c_nbnxnGpuNumClusterPerSupercluster * CL_SIZE)
#   else
__local float2* ljcpib      = (__local float2*)(LOCAL_OFFSET);
#       undef LOCAL_OFFSET
#       define LOCAL_OFFSET (ljcpib + c_nbnxnGpuNumClusterPerSupercluster * CL_SIZE)
#   endif
```

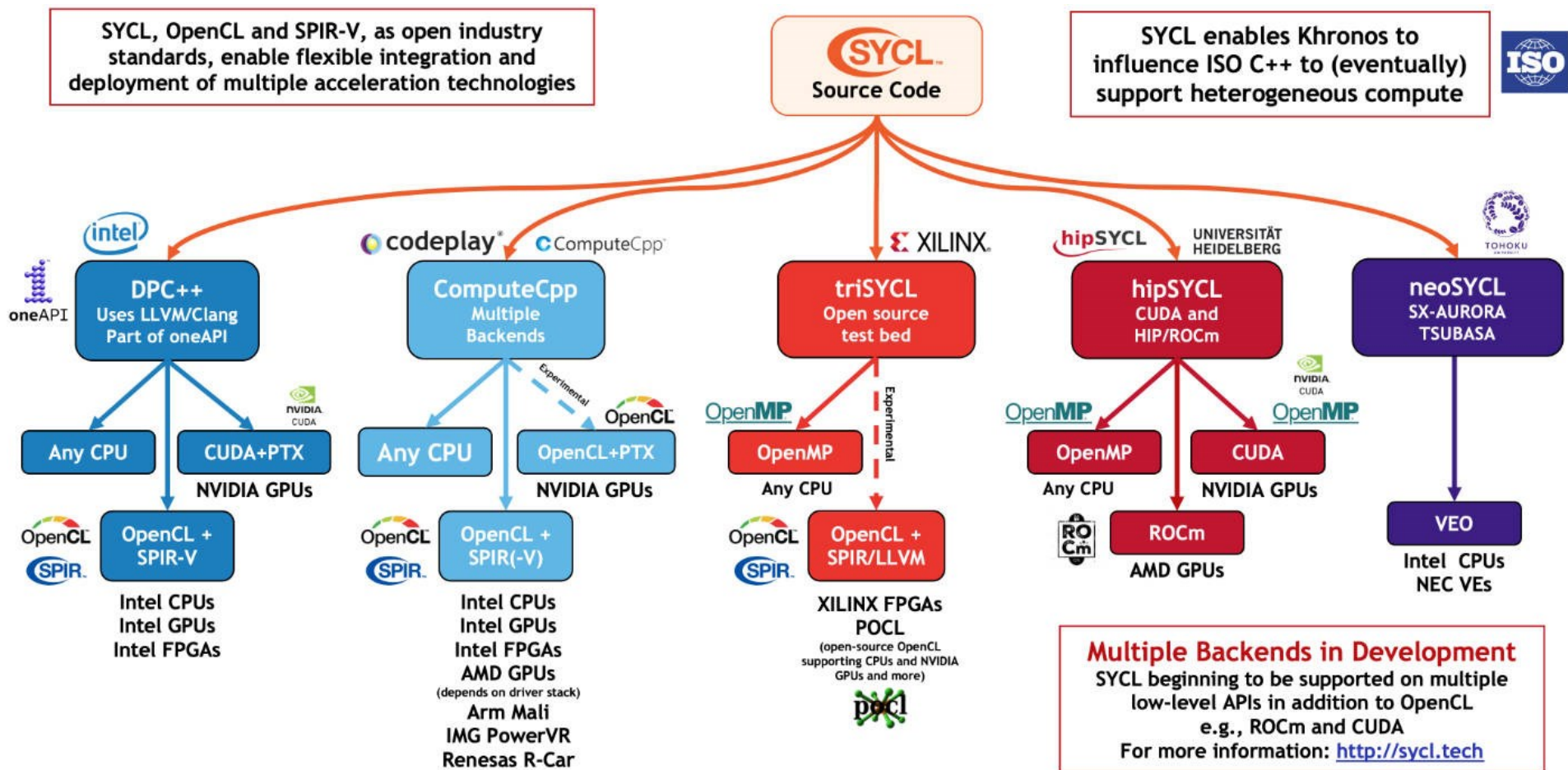
```
auto sm_atomTypeI = [&]() {
    if constexpr (!props.vdwComb)
    {
        return sycl::accessor<int, 2, mode::read_write, target::local>(
            sycl::range<2>(c_nbnxnGpuNumClusterPerSupercluster, c_clSize), cgh);
    }
    else { return nullptr; }
}();
```



```
auto sm_ljCombI = [&]() {
    if constexpr (props.vdwComb)
    {
        return sycl::local_accessor<Float2, 2, mode::read_write, target::local>(
            sycl::range<2>(c_nbnxnGpuNumClusterPerSupercluster, c_clSize), cgh);
    }
    else { return nullptr; }
}();
```



# SYCL beyond oneAPI



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

# 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
    - HIPSYCL\_EXT\_QUEUE\_WAIT\_LIST
  - Fix a few bugs not triggered with oneAPI
  - CMake scripting
  - Kernel optimizations ported from OpenCL

# Portability in practice: results

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

# Portability in practice: FFT

- 3D real-to-complex, forward and backward FFT
- Intel GPUs: use MKL
  - Had issues in oneAPI 2021.x
- AMD GPUs: use rocFFT via `HIPSYCL_EXT_ENQUEUE_CUSTOM_OPERATION`
- NVIDIA GPUs: not implemented, but can use cuFFT

# Miscellaneous

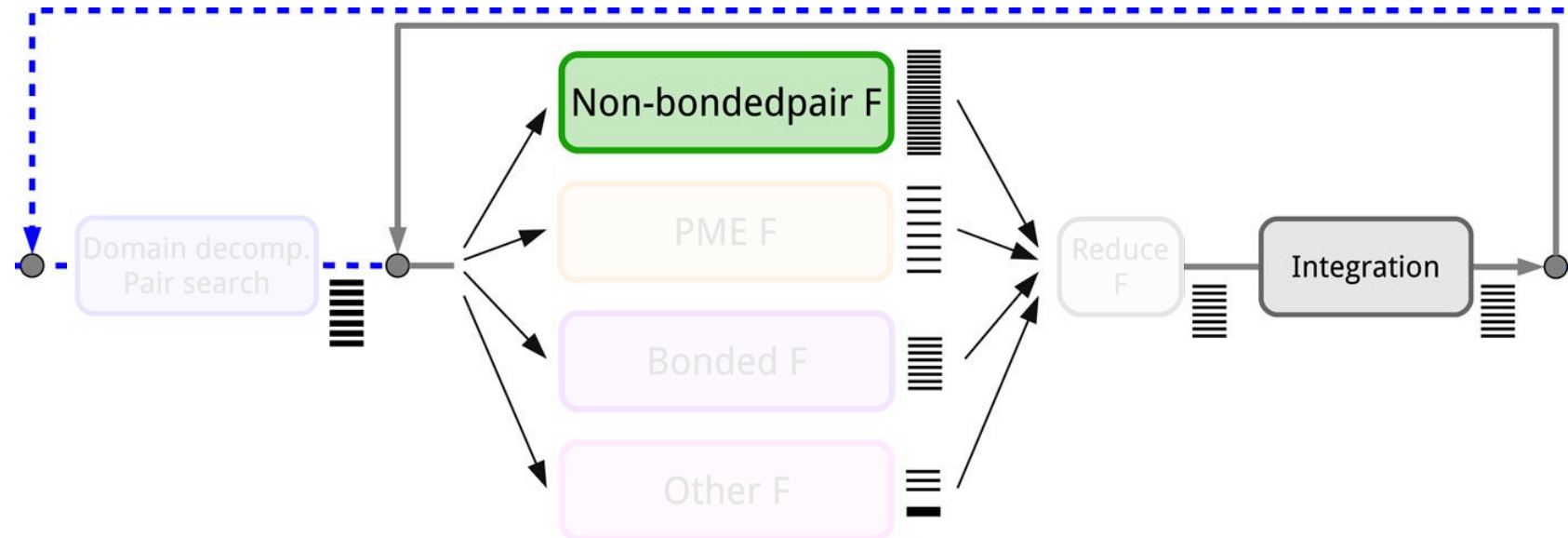
- Using existing profiling tools is great
  - E.g., NVIDIA Compute Sanitizer
- Compilation is slow
  - Even in runtime, thanks to JIT
  - Especially with 168 templated kernel flavors in a single file
    - `-fsycl-code-split=per_kernel`
- SYCL is relatively young and unstable
  - New and exciting features added
  - Things get deprecated and removed
    - Code written for oneAPI 2021.3 (Jul 2021) might not work with 2022.0 (Jan 2022)
      - Easy to fix
  - Things get broken

# SYCL in GROMACS: milestones

- September 2020: SYCL porting started, targeting Intel GPUs
- January 2021: “2021-sycl” release with basic functionality for Intel GPUs
- September 2021: SYCL selected for future AMD GPU support
- February 2022: SYCL is part of the mainline release, supports most GPU offload features, and shown to work on Intel, AMD, and NVIDIA GPUs

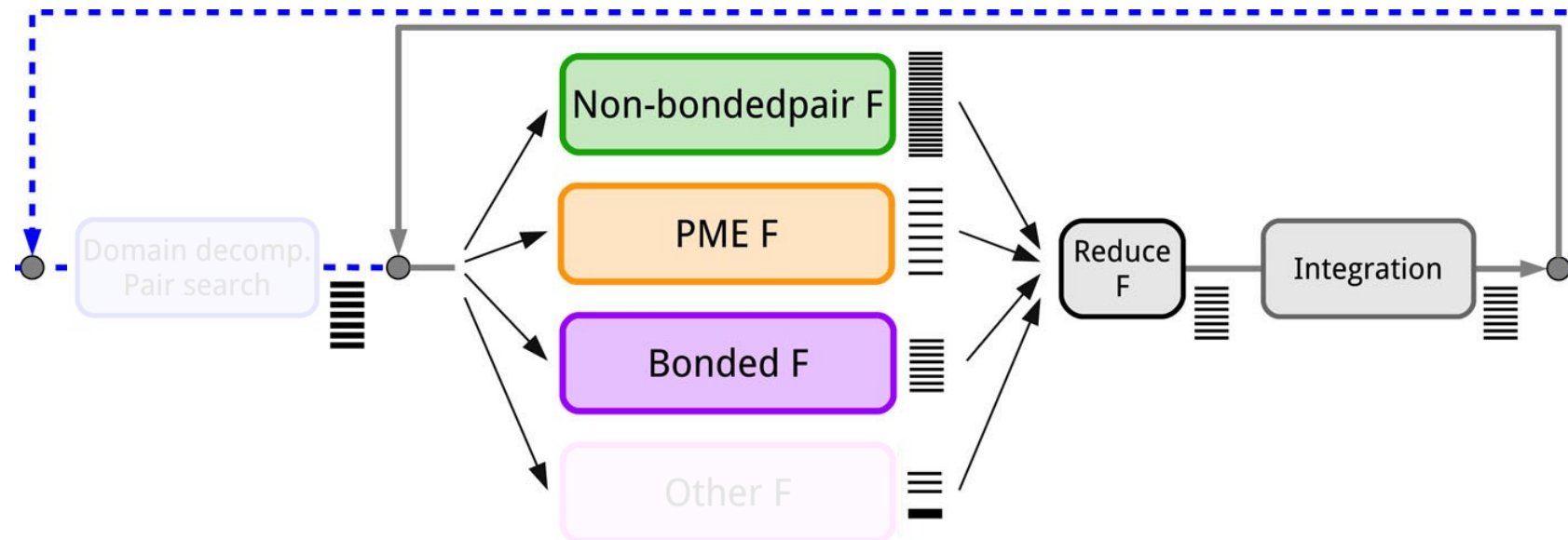
# SYCL support in GROMACS 2021

- Implementation: DPC++
- Hardware: Intel GPUs
- Offloaded operations: non-bonded forces, integration



# SYCL support in GROMACS 2022

- Implementation: DPC++ and [hipSYCL](#)
- Hardware: Intel and [AMD](#) GPUs; [NVIDIA](#) confirmed to work
- Offloaded operations: non-bonded forces, integration [with constraints](#), GPU update helpers, PME





# GPU feature support in GROMACS 2022

To be released February 2022



Device detection	✓	✓	✓
Non-bonded	✓	✓	✓
PME	✓	✓	✓
Update (integration)	✓	X	✓
Bonded	✓	X	X
Direct GPU-GPU comm	✓	X	X

# Conclusions

- “Write once, run anywhere” mostly works
  - Trivial changes to support all three major vendors with oneAPI and hipSYCL
- But running fast is neither easy
  - Still need vendor-specific code branches to get high performance
- ... nor guaranteed
  - On par with OpenCL with oneAPI DPC++, even faster when using LevelZero
  - Occasional large regressions with hipSYCL
- 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://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](mailto:andrey.alekseenko@scilifelab.se)**