Towards Performance Portability in NAMD with oneAPI

Tareq Malas, Intel Corporation
Jaemin Choi, University of Illinois at Urbana-Champaign

Programming Challenges

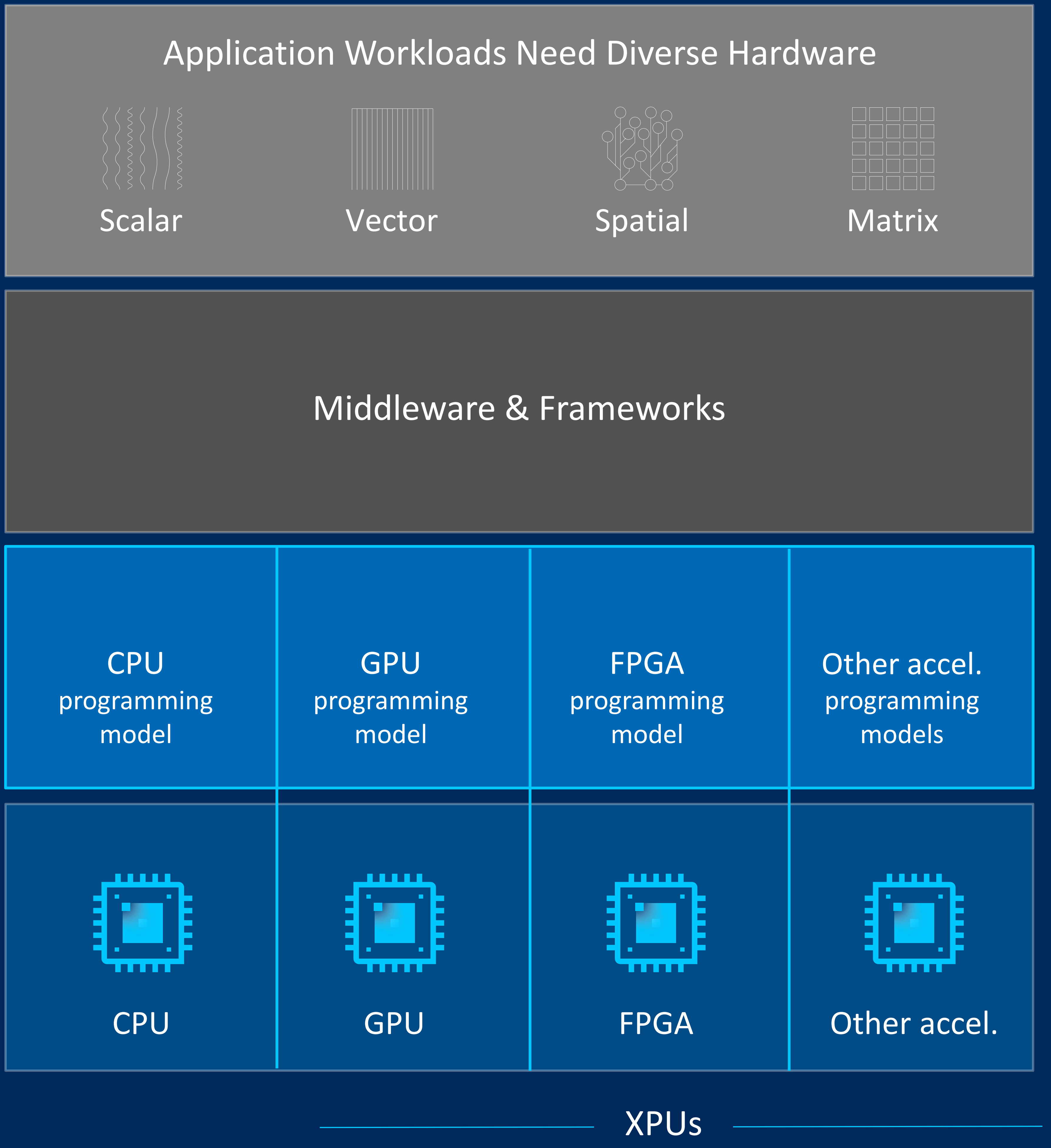
for Multiple Architectures

Growth in specialized workloads

Variety of data-centric hardware required

Separate programming models and toolchains for each architecture are required today

Software development complexity limits freedom of architectural choice



oneAPI

One Programming Model for Multiple Architectures and Vendors

Freedom to Make Your Best Choice

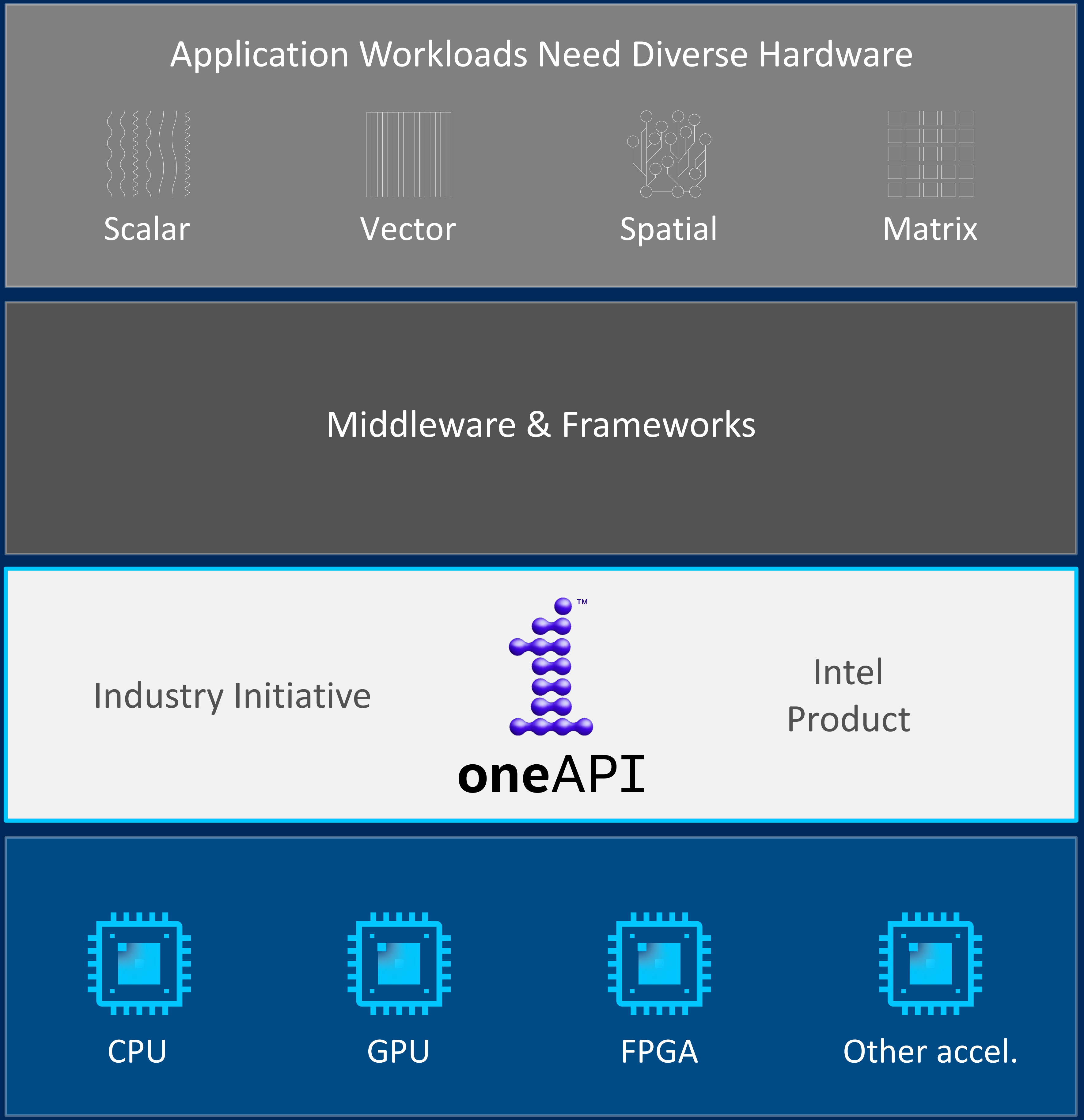
- Choose the best accelerated technology. The software doesn't decide for you

Realize all the Hardware Value

- Performance across CPU, GPUs, FPGAs, and other accelerators

Develop & Deploy Software with Peace of Mind

- Open industry standards provide a safe, clear path to the future
- Compatible with existing languages and programming models including C++, Python, SYCL, OpenMP, Fortran, and MPI

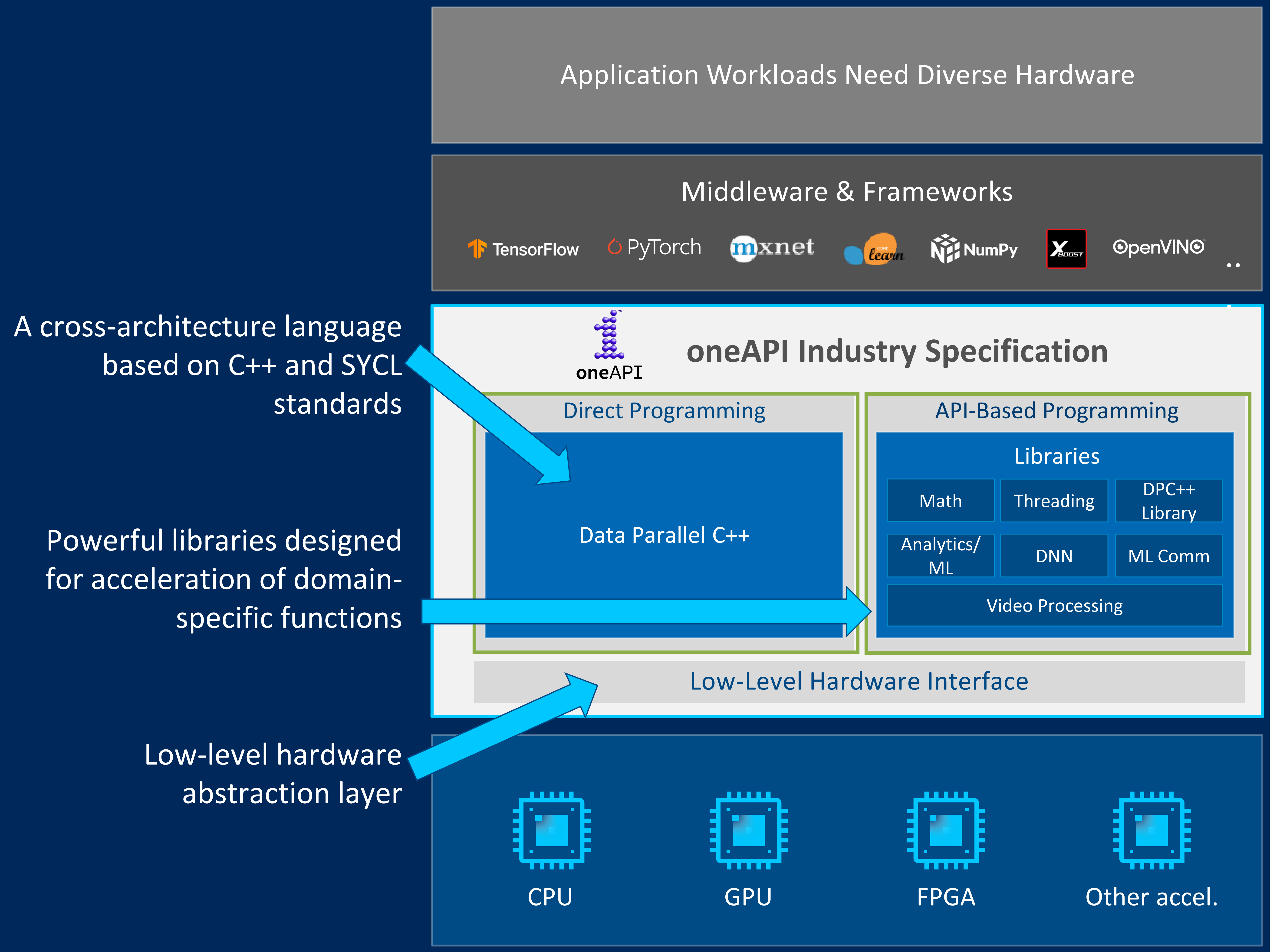


oneAPI Industry Initiative

Break the Chains of Proprietary Lock-in

Open to promote community and industry collaboration

Enables code reuse across architectures and vendors


Application Workloads Need Diverse Hardware
Middleware & Frameworks
TensorFlow
PyTorch
mxnet
learn
NumPy
XGBoost
OpenVINO
oneAPI
oneAPI Industry Specification
Direct Programming
Data Parallel C++
API-Based Programming
Libraries
Math
Threading
DPC++ Library
Analytics/ML
DNN
ML Comm
Video Processing
Low-Level Hardware Interface
CPU
GPU
FPGA
Other accel.
A cross-architecture language based on C++ and SYCL standards
Powerful libraries designed for acceleration of domain-specific functions
Low-level hardware abstraction layer

oneAPI

The productive, smart path to freedom for accelerated computing from the economic and technical burdens of proprietary programming models

Visit oneapi.com for more details

intel.

Why is NAMD adopting oneAPI?

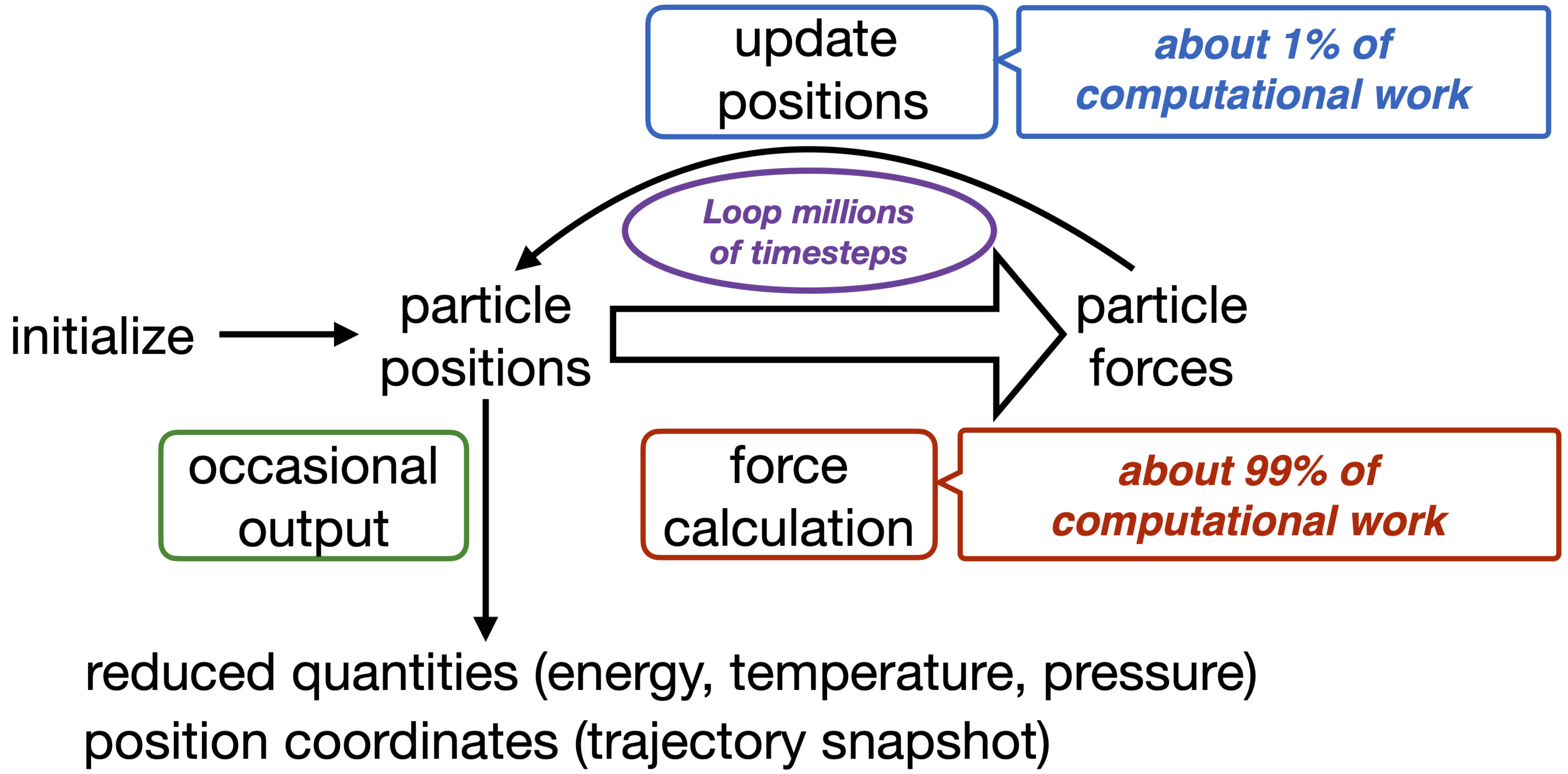
- Support upcoming exascale computers: ANL Aurora (Intel)
- oneAPI / DPC++ provides advantages
  - Modern C++ interface to GPU devices
  - Host-side code is **much** simpler than OpenCL
  - Same data structure definitions for both host and device
  - DPC++ incorporates open-standard SYCL with community extensions
  - Code portability across various hardware targets: CPU, GPU, FPGA

NAMD execution flow

force calculation

- 90%: Non-bonded forces, short-range cutoff
- 5%: Long-range electrostatics, gridded (e.g. PME)
- 2%: Bonded forces (bonds, angles, etc.)
- 2%: Correction for excluded interactions
- 1%:Integration, constraints, thermostat, barostat

update coordinates

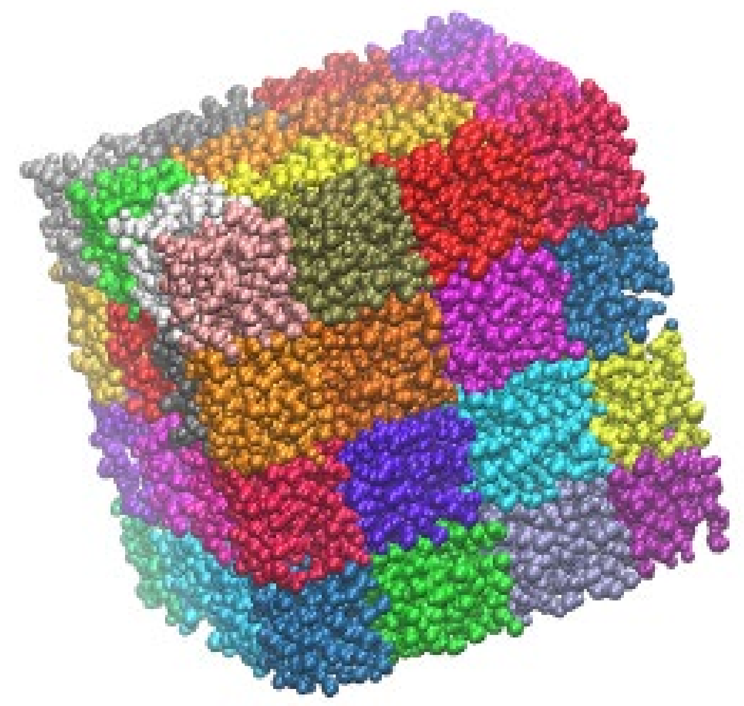


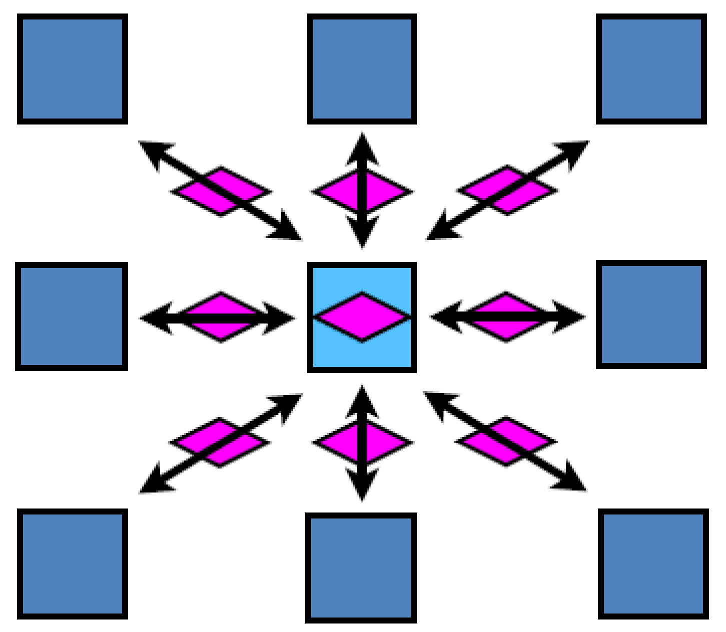
update positions
about 1% of computational work
Loop millions of timesteps
initialize
particle positions
particle forces
occasional output
force calculation
about 99% of computational work
reduced quantities (energy, temperature, pressure)
position coordinates (trajectory snapshot)

Improve Parallelism: Decompose Data **and** Work

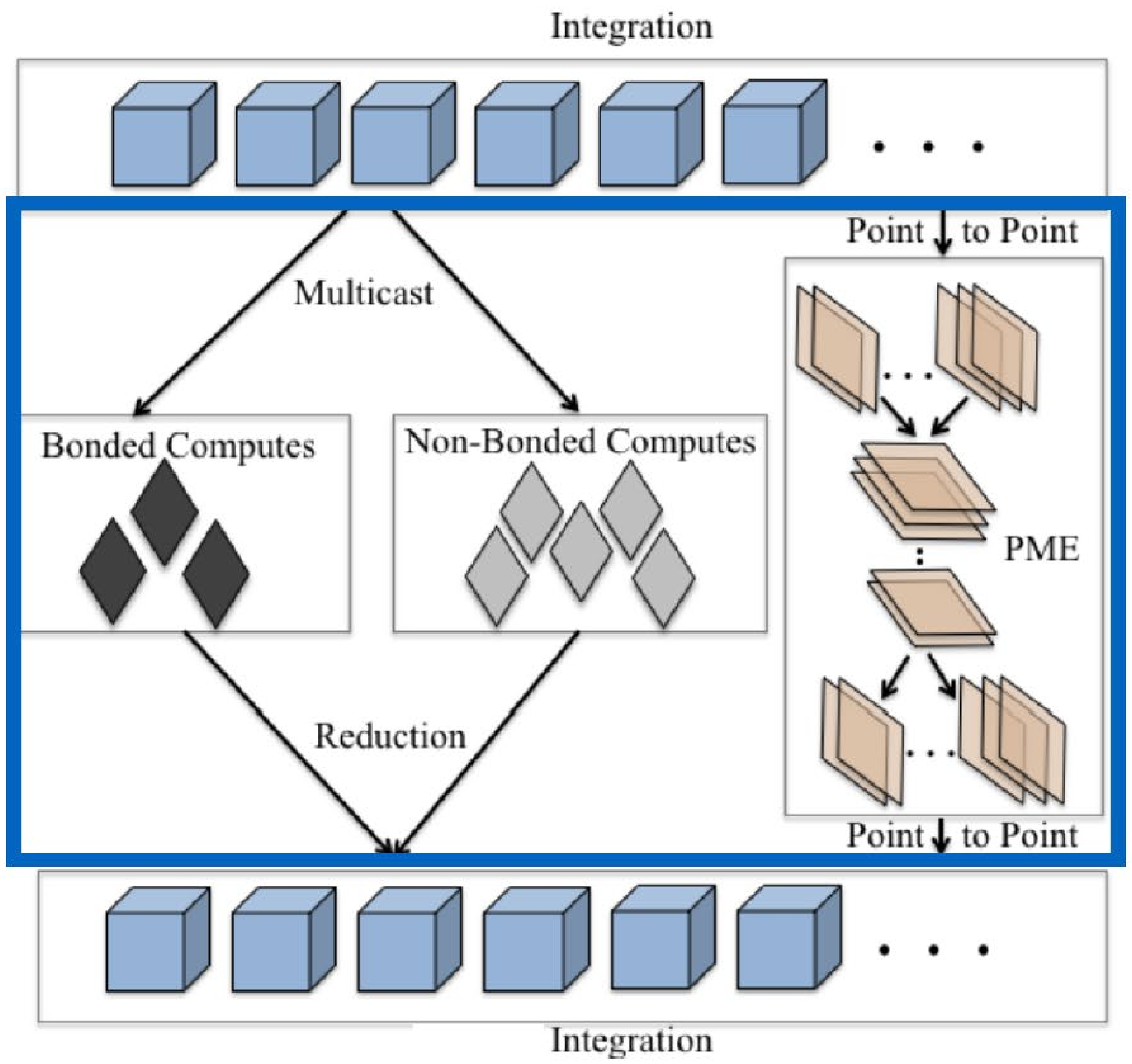
Kale et al., *J. Comp. Phys.* 151:283-312, 1999

- Atoms are decomposed into fixed volume **patches** within the system
- Forces that move atoms are calculated in parallel at each step between adjacent patches
- Work decomposition into compute objects creates much greater amount of parallelization, facilitates measurement-based load balancing with Charm++
- Migrate atoms to adjacent patches, updating domain decomposition after every **cycle** (e.g., 20 steps)


Spatial decomposition of atoms into patches


Work decomposition of patch interactions

NAMD Decomposes Force Terms into Fine-grained Objects for Scalability



Offload forces to GPU

NAMD has a **LOT** of CUDA code

- Start oneAPI / DPC++ porting with **stable code base** (version 2.14)
- NAMD 2.14 contains:
  - 38 C/h source files for the CUDA implementation (16K lines)
  - 8 *.cu files (6.8K lines)
- For example, porting the non-bonded force term to DPC++
  - Most computationally intensive part of the overall force calculation
  - 13 files (3 .cu, 4 .C, 6 .h), 7K lines
  - 11 kernels

National Institutes of Health

How does DPC++ differ from CUDA?

- Can provide what each thread will execute in parallel as a regular C++ function
- Uses C++ exceptions to catch errors (try-catch block)
- Memory management & transfers
  - Asynchronous by default
  - Associated with a SYCL queue (including allocations/frees)
- Shared memory allocations → local memory accessors (created before kernel invocation)
- Does not assume a certain SIMD width (warp and sub-group)
  - Should generalize warp-based mechanisms
  - Can enforce a sub-group size with [[reqd_sub_group_size(SIZE)]]

National Institutes of Health

Accelerated Development with DPCT

- Utilized Intel® DPC++ Compatibility Tool (DPCT) to accelerate code development
- Started with porting the CUDA implementation
  - DPCT saves > 80% of code porting effort
  - For example, threadIdx.x → ndItem.get_local_id(2)
- Provides a good source to practice DPC++ syntax

Porting Strategy

- We used a divide-and-conquer strategy, by using preprocessor switches to decouple the components in the CUDA code
  - Significantly reduces development and debugging complexity
- Separated components include
  - Non-bonded force & device utilities
  - Bonded force
  - PME
- Utilized oneDPL to use C++17 parallel STL sort and scan operations on the offload device

National Institutes of Health

DPC++ Offloading of the Force Computations

- Successful DPC++ offloading of NAMD to:
  - Intel® Xeon® CPU
  - Intel Gen9 integrated graphics
  - Intel DG1 and ATS discrete graphics
- Enabled multi-GPU and multi-node scalability with DPC++
- Includes implementation of DPC++ offload code management in NAMD
  - Interface with Charm++
  - Perform data management (transfer and storage)
  - Multiple CPU threads offloading to multiple DPC++ devices
- Validated benchmarks: Tiny (512 atoms), ApoA1 (90K), F1-ATPase (328K), STMV (1M)

Offloading PME with DPC++

- Completed porting of both PME code paths
  - PMEOffload (Jim), usePMECUDA (Antti-Pekka)
  - First pass with DPCT, manual updates for warp intrinsics & atomics
  - Replaced cuFFT with oneMKL FFT
- Validated correctness with single GPU (Gen9, ATS) and CPU offload

National Institutes of Health

DPC++ Improves Vectorization

- Using flexible vector width optimization towards performance portability to various architectures
- Changed use of CUDA warp primitives to generalized code supporting DPC++ sub-groups for efficient vector computation on different target architectures

Future Plans

- Make the DPC++ implementation available to NAMD community
    - Merge into the main public repository – targeting end-of-year
- Port NAMD GPU-resident code path (NAMD 3.0) to DPC++
- Use Intel® Vtune Profiler and Intel® Advisor tools to continuously optimize NAMD DPC++ for performance on Aurora supercomputer
- Experiment with NAMD DPC++ on NVIDIA and AMD GPUs

Debugging Challenges

- Chasing bugs in ported large applications from CUDA to DPC++ can be involved
  - Especially when dealing with large irregular arrays of structures
  - Large arrays may be pipelined to multiple kernels and code crashes at later stage when numbers become far from the expected value (e.g. NaNs)
  - Sometimes we are porting a complex application outside of our domain of expertise

National Institutes of Health

Proposed Solution

- Code porting mostly involves changing the syntax and library calls
  - All/most of the algorithm and result remain the same
  - Most/all non kernels code remains intact
- Add utilities to capture kernels' input/output across languages (DPC++, CUDA)
  - Write to a file the input/output data of the kernels in reference language
  - Read the data files in the development code and compare the arrays
- Results
  - Developed easy to add macros around the kernel call
  - Allows the developer to capture the first difference location in code and data

Acknowledgments

- David Hardy for leading the development efforts
- NAMD development is funded by NIH P41-GM104601
- Grant from Intel for COVID research software tool development
- UIUC with Intel has established oneAPI academic Center of Excellence
- Intel engineers: Mike Brown, Alex Wells, and Robert Cohn