Neon: A Multi-GPU Programming Model for Grid-based Computations

Massimiliano Meneghin[1†], Ahmed H. Mahmoud[1 2†], Pradeep Kumar Jayaraman[1], Nigel J. W. Morris[1]
[1]Autodesk Research, Canada, [2]University of California, Davis
Email: {massimiliano.meneghin, ahmed.mahmoud, pradeep.kumar.jayaraman, nigel.morris}@autodesk.com

***Abstract*—We present Neon, a new programming model for grid-based computation with an intuitive, easy-to-use interface that allows domain experts to take full advantage of single-node multi-GPU systems. Neon decouples data structure from computation and back end configurations, allowing the same user code to operate on a variety of data structures and devices. Neon relies on a set of hierarchical abstractions that allow the user to write their applications as if they were sequential applications, while the runtime handles distribution across multiple GPUs and performs optimizations such as overlapping computation and communication without user intervention. We evaluate our programming model on several applications: a Lattice Boltzmann fluid solver, a finite-difference Poisson solver and a finite-element linear elastic solver. We show that these applications can be implemented concisely and scale well with the number of GPUs—achieving more than 99% of ideal efficiency.**

I. Introduction

Structured grids are employed in various applications including thermal, mechanical and fluid simulations, computer graphics, visual effects production and medical imaging. Many algorithms—including spatial interpolations, resampling, convolutions, and numerical schemes to discretize differential equations—are simpler to realize on structured grids. Despite their simplicity and potential for cache-friendly data layouts, fine resolution grid-based data incur a large compute and memory cost, requiring high-performance computing clusters and distributed programming models for scalability.

GPUs are increasingly being employed to scale high-performance applications like machine learning, image processing, and scientific visualization. Domain experts in scientific computing could greatly benefit from multi-GPU acceleration of their programs. However, designing, implementing, and debugging parallel algorithms to leverage multiple GPUs requires mastery of multi-GPU programming concepts—expert knowledge of new languages like CUDA or OpenCL, understanding of parallel programming models, handling asynchronous computations, communications and memory transfer.

Consider designing a typical map-stencil pattern on a simple dense domain e.g., an AXPY followed by a Laplacian filter. Its execution workflow is shown in Fig. 1, from a naïve version to an optimized one that aims at maximizing the overlap between computation and communication (OCC). The more aggressive an OCC optimization is, the more complex the code becomes, and a deeper knowledge of multi-GPU communication is needed. The complexity increases even further when considering sparse volumetric representations because of the lack of a data structure abstraction that supports a general and portable definition for OCC optimizations.

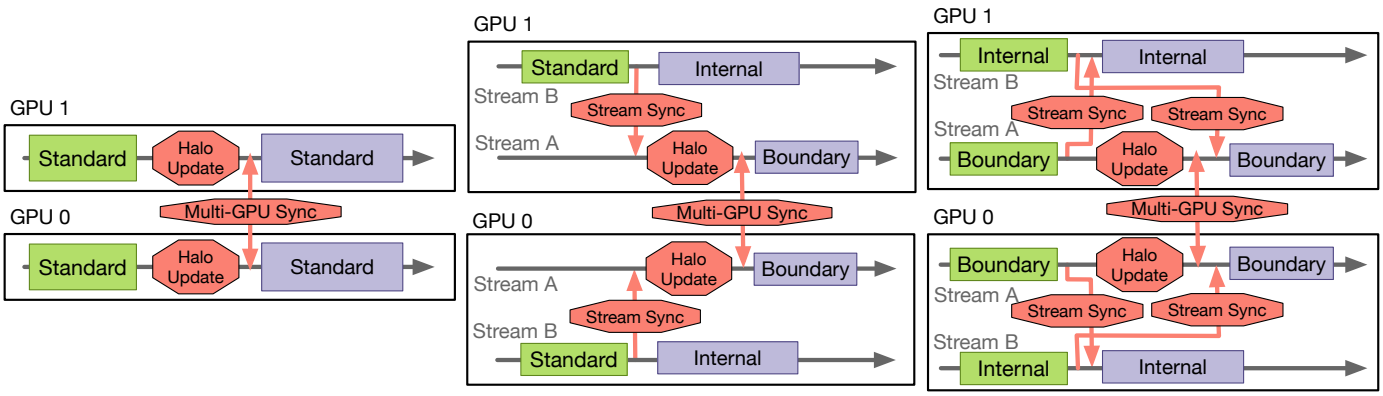
GPU-based frameworks like TensorFlow [1] for machine learning, Halide [2] for image processing, and Hadoop [3] for big-data processing are some examples where programming multi-GPU and distributed systems have been made easier for specific applications. It has been argued that these frameworks have led to rapid scientific and technical advances in many fields [4]. We aim to bring similar benefits to large-scale grid-based numerical problems while delivering excellent efficiency by leveraging automatic OCC optimizations.

We present Neon—a programming model and its C++ implementation that enables scaling of grid-based numerical problems on single-node multi-GPU systems, without sacrificing ease of programming or requiring parallel programming knowledge from the user. Using Neon, we are able to achieve more than 99% ideal efficiency on the analyzed applications thanks to the automatic OCC optimization. Neon follows a domain specific approach—it provides an intuitive API to represent numerical fields over a regularly discretized 3D domain. As in most engineering problems, the domain is free-form (i.e., not a cubic). We also provide options for dense and sparse representations—both optimized for load balance and minimization of GPU-GPU transfer overhead.

Neon's computational model derives from the parallel skeleton tradition [5] where users describe their applications using a set of predefined parallel constructs. By design, any Neon application is described with respect to a back end (CPU or GPU), the number of available resources (GPUs), a grid data structure (dense or sparse), layout (Structure-of-Arrays or Array-of-Structures) and memory properties (alignment, padding, type of allocator). More importantly, all parameters can be easily switched without altering application-specific code. Unlike other skeleton models, Neon treats grid data structures as first-class citizens which allows for many optimizations. In this paper, we make the following contributions:

- A programming model that abstracts multi-GPU complexity and exposes an intuitive interface to users. We present a method to automatically infer the data dependency graph and introduce OCC optimizations from a sequential code.
- A C++ implementation of our programming model with an easy-to-use interface for grid-based computations. We demonstrate how the grid data structure (dense, sparse) and

[†]Joint first author



(a) Unoptimized (b) Overlap stencil with memory (c) Overlap map and stencil with memory

Fig. 1: A schematic implementation of a map operation (green) followed by a stencil computation (purple) using two GPUs with different levels of optimization and complexity: a) The unoptimized implementation does not consider OCC by setting a global synchronization barrier before the halo update, b) Overlapping part of the stencil operation with memory transfer is a common technique to improve performance, c) Further overlapping between computation and communication is achieved by splitting the map operation and initiating memory transfer right after the boundary map computation, potentially leading to better performance.

back end (multi-core, single GPU, and single-node multi-GPU) can be changed with no modification to user-code.
- Benchmarks on an 8-GPU system demonstrating excellent scaling performance on three real-world applications: a Lattice-Boltzmann fluid solver, a finite-difference Poisson solver, and a finite-element linear elastic structural solver.

We restrict the scope of our implementation of the Neon programming model to single node multi-GPU and without support for out-of-core operations. Our current implementation and explanation of the system is based on CUDA for simplicity and only targets Nvidia GPUs. However, the principles of the system are generic and allow similar implementations using other frameworks and APIs like OpenCL or Vulkan.

II. Related Work

Since our goal is towards a high-level, generic abstraction for grid-based computation, we only discuss related work that allows developers to write simple code for their *full* application, while relying on a framework for optimization across multiple GPUs.

a) Generic Frameworks: Modern generic parallel computing frameworks capitalize on the so-called pattern-based (or skeleton-based) programming model to offer a high-level parallel computing abstraction without sacrificing high performance. Complex applications are built on top of these patterns by using them as building blocks, while the low-level details are hidden from the user. With their concise semantics and clear functionality, frameworks using parallel patterns strike a good balance between simplicity, high performance, and portability [6], [7], [8], [9]. Since these frameworks target generic applications, the user has to implement and optimize their data structure for grid-based applications using low-level primitives (e.g., 1D arrays) offered by the framework—a time-consuming task especially for sparse domains. Neon follows a pattern-based programming model, while treating the grid data structure as a first-class citizen which allows for optimizations that are very challenging using low-level primitives.

b) Domain-specific Frameworks: Many grid-based application-specific frameworks have been proposed to hide the complexity of GPU programming while offering intuitive abstraction for the application developers [10], [11], [12]. Grid-based computation can benefit from a higher-level abstraction where developers can write all their applications in a single framework and, more importantly, share optimization techniques across different applications.

Taichi [13] and OPS [14] are the most closely related works to ours, and develop a domain-specific framework for grid-based computation. Taichi is a Python framework that relies on just-in-time compilation to offload compute-intensive tasks to multi-core CPUs or a single GPU. Similarly to Neon, Taichi makes it possible to interchange the data structure without a change in the computation code. However, extending Taichi to multi-GPU requires extensive user engagement to control each GPU's memory. OPS is a domain-specific language and active library embedded into C/C++/Fortran for expressing computations on parallel multi-block structured mesh. It allows for a number of automated communication-pattern optimizations on a cluster of GPUs or multi-core CPUs using source-to-source translation, delayed execution, and dependency analysis. Both Neon and OPS relieve the user from low-level optimization by performing dependency analysis and automatic partitioning of the domain into computational resources. However, Neon is more extensible by allowing users to experiment with different data structures while OPS uses only a multi-block data structure with user-managed inter-block halo exchanges.

```
Neon::Int3 dim(128, 128, 1);
std::vector<int> gpuIDs{0, 1, 2};
Neon::Backend backend(Neon::Backend::CUDA, gpuIDs);
// Alternatively, user can switch to OpenMP back end
// Neon::Backend backend(Neon::Backend::OpenMP);
Neon::Grid grid = Neon::Grid(backend, dim,
        // Specify active grid cells
        [&](const Neon::Int3& idx) {
            int radius = 5*5;
            return idx.x*idx.x + idx.y*idx.y <=radius;
        });
int cardinality = 3;
float outsideDomainValue = 0;
Neon::Field velocity = grid.newField<float>(
                      cardinality, outsideDomainValue);
```

Listing 1: Example of creating 2D circular domain.

III. Neon Programming Model

At a high level, our programming model decouples the data structure from the computation performed on it. The user specifies the domain properties (e.g., size, sparsity pattern) and then defines computation on top of it. The computation is parameterized by the domain so that the user can change the domain properties without changing the computation code. We capitalize on the data-parallel nature of numerical computation. The user only focuses on specifying what operations to apply on an *active* grid cell while Neon takes care of applying these operations on all active grid cells and performs optimizations to achieve high performance.

a) Domain: Computational domains are built from two components: Grid and Field. Grid is the blueprint for organizing the computational layout. It defines the extent of the domain, the sparsity pattern, and the runtime back end. Field stores the physical quantities needed by the application and is defined by its properties (e.g., cardinality, data type, and data layout) and the Grid type that creates it. Listing 1 shows how to define a simple Grid and Field for a 2D domain.

b) Computation: We define three building blocks in our programming model that can be used to write a wide range of numerical applications. All our building blocks accept Fields as input and exhibit different communication patterns, thereby requiring different types of optimization for high performance.

Map Operation (`Neon::MapOp`): It updates the data associated with a grid cell in a Field using data from the corresponding grid cells in other input Fields, potentially transformed by user-defined operations. Concretely, map operations are defined as follows where F_* are all Fields of size K, and $op(\cdot)$ is a user-defined operation:

$$F_{out}[i] \leftarrow op(F_1[i], F_2[i], \dots), \forall i \in [0, K).$$

Examples of Map operations include AXPY, copying a Field to another, and scaling a Field by a constant.

Stencil Operation (`Neon::StencilOp`): It allows a grid cell to gather information from surrounding grid cells using a user-defined stencil. Similar to Map, the user can define custom operations and specify how the final results are aggregated. Formally, stencil operations are defined as follows where $op(\cdot)$ is user-defined operation and $\mathcal{N}(i)$ is the set of stencil neighbour grid cells around cell i:

$$F_{out}[i] \leftarrow op_{j \in \mathcal{N}(i)}(F_1[j], F_2[j], \dots), \forall i \in [0, K).$$

Reduce Operation (`Neon::ReduceOp`): It applies a user-defined binary and associative operation on the input Fields and reduces them to a single value. Examples include dot product and computing the L^2 norm of a Field. The reduce operation can be defined as

$$out \leftarrow op(\{F_1[i], F_2[i], \dots \mid \forall i \in [0, K)\}).$$

While these three building blocks are simple, they are powerful enough to write a wide range of computations such as solving linear systems, eigenvalue problems and almost all the functions found in BLAS [15]. The major difference is how Neon defines matrix operations. While BLAS aims at general-purpose matrix operations, the majority of grid-based numerical computations define their matrix operations as a stencil applied on a well-defined neighborhood around each grid cell. Thus, Neon performs such operations in a matrix-free fashion, avoiding the costly assembly step [16]. One limitation of this approach is that certain matrix operations (e.g., matrix factorization) must be user-managed.

An example of using these building blocks is shown in Listing 2 for creating a Laplacian stencil. Neon provides an opaque data type called Container to wrap one or more building blocks. The building block type is defined on the Field itself (line 4 and 5) using the Loader data type (Section IV-B). Containers allow the user to create complex operations by combining multiple building blocks. The actual functionality is implemented as a lambda function (lines 6–17).

In order to construct a useful application, the user defines a set of sequential steps, where each step is wrapped by a Container. Neon provides a Skeleton data type that takes the sequence of Containers, along with the desired back end to run them, and is responsible for exploiting parallelism when possible (Section V-B). This is achieved by automatically building a data dependency graph (Section V-A) from the Containers. Listing 3 shows a pseudo code implementation of a conjugate gradient solver [17] using the Neon programming model. Along with the user-defined building blocks, Neon also offers a set of well-optimized standard BLAS operations (e.g., dot product) with a unified interface for different grid types to facilitate rapid prototyping.

IV. Neon Framework

In this section, we dive into the design of Neon. We aim to develop a domain-specific programming model with the following high-level design goals:

Simplicity: We aim to simplify the user experience for programming grid-based multi-GPU applications. Neon's target is to provide a programming model that allows users to write applications as sequential code while their execution runs on a multi-GPU back end.

```
Neon::Container LaplacianStencil(const Field& input,
                                 Field& output) {
  return input.newContainer([&](Neon::Loader& loader){
    auto& inp = loader.load(input, Neon::StencilOp);
    auto& out = loader.load(output, Neon::MapOp);
    return [=](const Field::Index& idx){
      Field::Index direction;
      for (int c = 0; c < inp.cardinality(); c++) {
         Field::Type sum = 0;
         direction = {1,0,0}; //+x
         sum += inp.neighbourValue(idx, direction, c);
         direction = {-1,0,0}; //-x
         sum += inp.neighbourValue(idx, direction, c);
         // ..... similarly for -y, +y, -z, and +z

         out(idx, c) = (-sum + 6.0 * inp(idx, c));
      }};});
}
```

Listing 2: User implementation of 7-point stencil operation in Neon used to compute the Laplacian.

```
Neon::Backend backend;
Neon::Grid grid;
Neon::Field p, s, x /*unknown*/, r /*residual*/;
Neon::Scalar rr0, rr, ps;
// initialize backend, grid, fields,
// and initial residual (rr)
//...

// Create a skeleton from a list of Containers
std::vector<Container> CG {
    LaplacianStencil(p, s),  // s := A*p
    grid.Dot(p, s, ps),      // ps := <p,s>
    UpdateX(rr, ps, p, x),   // x := (rr/ps)*p + x
    UpdateR(rr, ps, s, r),   // r := (-rr/ps)*s + r
    Residual(r, r, rr, rr0), // rr0 := rr
                             // rr := <r,r>
    UpdateP(rr, rr0, r, p)}; // p := r + (rr/rr0)*p

Neon::Skeleton skeleton(backend, CG);
while(...) { skeleton.run();}
backend.sync();
```

Listing 3: Pseudo code of the conjugate gradient algorithm using Neon programming model.

Performance: Without sacrificing API simplicity, we aim to achieve performance that closely matches hardwired application-specific code. This is done by automatically partitioning and distributing user data, minimizing the parallelization overhead (communication and synchronizations), and ensuring load balancing.

Decouple data structures from computation: Performance depends heavily on the underlying grid data structure. Thus, Neon separates the data structure from operations applied on the data structure. This makes it possible to test different data structures' performance without changing user code.

Portability: While we aim to provide high performance on a multi-GPU back end, debugging of a user's application on a serial back end is common practice. Neon allows portability between different back ends with a focus on high performance on the multi-GPU back end.

To reach these goals, we need to address four challenges:

Data Challenge: We need a generic multi-GPU data abstraction to distribute, manage, and access user data across the available GPUs. The abstraction must also determine data exchanges between GPUs to support the correct execution of stencil and reduce operations.

Kernel Challenge: We require a multi-GPU kernel mechanism that can transform user sequential kernel code into GPU-specific functions for the GPU-specific portion of data.

Dependency-graph Challenge: The system needs a mechanism to detect what data a multi-GPU kernel is using, its access type (read or write), and the compute pattern (`mapOp`, `stencilOp`, or `reduceOp`) in order to build a dependency graph of the application. We focus only on applications described as a user-defined sequence of functions—we do not support dynamic branching.

Orchestration Challenge: We have to define a correct and optimized execution strategy for the user-defined sequence of functions that compose the application. From a dependency graph and the kernel's computation pattern, the orchestration process must automatically reorganize the execution to include the necessary synchronizations and communication for the correct execution. Then optimizations that alleviate the parallelization overhead, like OCC must be incorporated.

While other tools like OPS [14] or Taichi [13] address some of the previous challenges by relying on a compiler, we designed Neon as a self-contained C++ library. Because of the limited run-time introspection mechanism in C++ the Dependency-graph challenge becomes a non trivial task. Nevertheless, the C++ language is widely used in the HPC community [18] and thus a C++ library is amenable to be integrated into existing HPC code bases.

We designed Neon by defining a hierarchy of abstraction layers. Each layer extends the previous one with semantically powerful mechanisms. At the lowest level, the Neon *System* (Section IV-A) provides an object-oriented and unified interface for any supported back end. The *Set* (Section IV-B) level adds mechanisms to address the *Kernel Challenge*, *Dependency-graph challenge* and partially address the *Data Challenge*. The *Domain* extends and automates the *Set* by providing domain-specific mechanisms for grid computation finalizing our solution for the *Data Challenge*. Finally, at the highest abstraction, the *Skeleton* (Section V) leverages mechanisms from all previous levels to solve the *Orchestration Challenge*. The end result is a programming model that efficiently hides all multi-GPU complexity.

A. The System Abstraction

The System abstraction shields the rest of Neon from architecture and hardware-specific mechanisms. It defines an

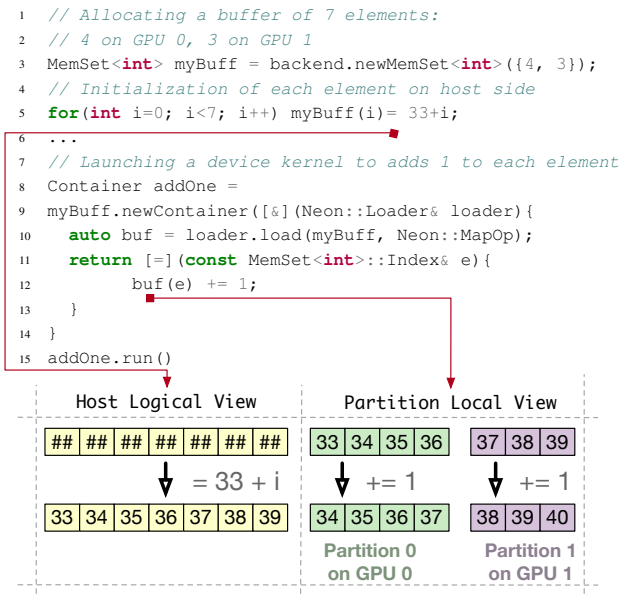


Fig. 2: In Neon Set abstraction, different data views are used: a continuous logical view for the host, and partitioned local view for the GPU. Operations on the data are defined using the Container with for-each interface (line 11).

object-oriented interface to manage resources and requires the following back end capabilities:

- *Memory Management.* This allows Neon to create device buffers and move data between devices or the host.
- *Queue-based Run-time Model.* Neon uses a queue-based model to abstract asynchronous kernels running on the same device. It is a generic model widely used at the hardware level e.g., in CUDA, *Streams* represent command queues, while *Events* are the mechanism to inject dependencies between different queues.
- *Lambda Functions*. Neon leverages the expressiveness of lambda functions to lessen the complexity of authoring multi-GPU applications.

Therefore, only the Neon System abstraction must be implemented to port Neon to a new accelerator that exposes the previous mechanisms. All the higher levels are not impacted.

We model single node, multi-core CPU systems with the same accelerator interface defined for GPUs. However we limit the system to only one kernel at the time.

B. The Set Abstraction

The Set abstraction offers solutions for both our *Data Challenge* and *Kernel Challenge* by defining rules to automatically transform a single sequential user kernel into a semantically equivalent multi-GPU kernel execution. Here we model a multi-GPU system by parametrizing all its mechanisms with respect to the available resources i.e., data and kernels are described as vectors where the i-th entry stores the information associated with the i-th GPU.

1) Multi-GPU Data: The Set abstraction defines a C++ template abstract interface for **Multi-GPU data** to manage data that is partitioned, distributed, and stored over the available GPUs. Any implementation of the Multi-GPU data interface will create a partition for each GPU and provide an interface to access it. However, the interface does not impose any restriction on the partition structure, layout or implementation.

In Neon, *MemSet* is the simplest example of an object implementing the Multi-GPU data interface. MemSet is one of Neon's memory allocators for multi-GPU buffers. Given allocation requests for each GPU, it allocates distributed buffers on the target GPUs, and optionally mirrors the set of buffers on the CPU (see example in Fig. 2). MemSet also creates a mapping from the host buffers to the GPU buffers and provides methods to transfer between the host and device. The actual allocations and memory updates are operated through the System abstraction. Note that MemSet does not handle automatic partitioning or load balancing; these mechanisms are introduced at the Domain abstraction level.

The Neon Multi-GPU data provides:

- *Host Logical View.* When accessed from the host side, a Multi-GPU data object exposes a contiguous logical view of its data. An example for MemSet is shown in Fig. 2, line 5 where we interact with the MemSet object as a contiguous buffer.
- *Partition Local View.* When queried by the accelerators, a Multi-GPU data object exposes an interface to read and write elements of any of its partitions. The interface is index-based, where both the index space for each data partition and the index type are accessible to Neon.

2) Containers - Kernels for Multi-GPU: In a multi-GPU environment, we are looking for a new kernel concept which we call *Container*. Its design goals are 1) to provide users with mechanisms to write a single code computing on Multi-GPU data, and 2) to enable the framework to generate GPU-specific versions of the code where Multi-GPU data objects are replaced with their local GPU partition. While this would be a trivial task with the support of a custom compiler, we only leverage standard C++ mechanisms that can generate code such as templates and lambda functions.

Fig. 2 shows an example of Container code. A Container is created from a Multi-GPU data object (`newContainer` method, line 9) which provides the information for the index space for each partition. The GPU-specific generated code is a lambda function, called *Compute Lambda*, which is written by the user and works on local partitions of Multi-GPU data objects captured by value (lines 11–13).

The signature of the Compute Lambda is constrained to one input parameter so that the system can easily generate template-based lambda executors at compile-time for both CUDA and OpenMP back ends. The input parameter is an index-based iterator provided by Multi-GPU data object used to create the Container. Defining the compute lambda with a single input parameter does not limit the flexibility of the approach as any required values for the kernel can be captured.

As a C++ library, Neon does not have any parsing capabilities to automatically detect what or how Multi-GPU data

are used by the Compute Lambda. Thus the Container design includes a loading process (Fig. 2, line 10) and a Loader object for the users to explicitly 1) extract the local view from the Multi-GPU data, and 2) express what type of computation the Multi-GPU data will be used for (`mapOp`, `stencilOp`, or `reduceOp`). The loading process and the generation of the Compute Lambda must be executed by the system for each GPU. Therefore, the two components are embedded into a second lambda function called *Loading Lambda* (Fig. 2, lines 9–14), which returns the generated Compute Lambda.

```
1 void Container::run(){
2     for(auto gpu_i : allGPUs) {
3         Loader loader_i(gpu_i);
4         auto compute_i = getComputeLambda(loader_i);
5         auto indexSpace_i = getIndexSpace(gpu_i);
6         gpu_i.execute(indexSpace_i, compute_i);
7 }}
```

Listing 4: Pseudo code of how Neon processes a Container for execution.

Listing 4 illustrates how Neon leverages the Container design for code execution as called in Fig. 2, line 15. Neon loops over all the available GPUs and starts by initializing a GPU-specific Loader object. The Loader is passed as input to the Loading Lambda to extract the generated Compute Lambda (line 4). The GPU-specific Loader object hides the SPMD nature of the Container, acting like the *rank* mechanism in MPI. Neon then computes the index space from the Multi-GPU data object that was used to create the Container (`myBuff` in Fig. 2, line 8) to determine the CUDA grid parameters and runs the lambda function on the target device.

3) Application Graph: The Container also solves the Dependency-graph Challenge by leveraging the Loader abstraction. As by design users explicitly extract the Multi-GPU data partition local view from a Loader object during the loading process, the Loader can store information about all the Multi-GPU data used in a Container. Therefore from a sequence of Containers, a data dependency graph can be defined. This capability is then leveraged by the Skeleton abstraction (see Section V).

4) A parametric run-time model: Finally the Set abstraction extends the queue-based run-time model defined by the System level to a multi-GPU environment. Following CUDA nomenclature, the Set level provides multi-GPU extension for CUDA Streams and CUDA Events. A multi-GPU Stream is simply a vector storing one CUDA Stream for each of the GPU. The same is for multi-GPU Events. At this abstraction level, users can manually manage multi-GPU Streams and multi-GPU Events to manage the execution of Containers, however higher levels in Neon will manage them automatically.

C. The Domain Abstraction

The Domain abstraction introduces the Grid, and Field models, which together allow grid-specific automation and optimizations. This level relies on the Set abstraction to define user computations but it also contributes to the last missing aspect of the *Data Challenge*—automatically managing GPU-GPU data transfers.

1) Grid: Grid is an abstraction for rectilinear domains and provides a blueprint for defining data on top of it. The abstraction automatically manages the partitioning of the rectilinear domain over the available GPUs. To efficiently handle stencil operations, the Grid defines a *data view model* that is inspired by MPI optimizations where cells are categorized by their data dependencies with respect to a stencil operation (Fig. 3). *Internal* cells (green) can be computed by accessing only the local partition data. *Boundary* cells (red) require access to the halo data received from neighbouring partitions. The *Standard* view is the union of internal and boundary cells. Neon determines which cells are boundary or internal based on the user-provided stencils at initialization. When launching a Container, a data view parameter specifies which *view* the Container should work on i.e., internal, boundary, or standard. As presented in the next session, this option enables a higher abstraction level in Neon to automatically inject essential scheduling optimizations for OCC.

2) Field: Field represents scalar or vector metadata associated with each cell of the Grid. The Field extends the Set abstraction Multi-GPU data interface (Section IV-B1) with domain-specific capabilities—the Partition view of the Field allows users to read and write cell metadata as well as to read metadata of neighbouring cells (Listing 2, lines 16 and line 11, respectively). For the Field abstraction, we follow the own-compute rule model and therefore neighbour cell metadata can not be modified. To support metadata access for cells that are not local to a Partition, the Field abstraction includes an explicit a halo coherency model—a *haloUpdate* asynchronous mechanism that updates halo data with respect to a remote Partition. The size of the halos are computed based on the union of all the stencils.

Neon provides two types of Grid representations: *dense*, where the entire domain is represented in memory, and *element-sparse*, where only cells of interest are stored with their connectivity table. The Grid API also provides a way to specify the layout for vector metadata by supporting Structure-of-Arrays (SoA) or Array-of-Structures (AoS) organizations without impacting the application code. Listing 2 showcases a 7-point stencil operation with Grids and Fields.

The Grid and Field define abstraction interfaces and do not impose any restriction on implementations. In particular, on modern shared-memory multi-GPU systems there are two main solutions to implement a halo coherency model:

- *Explicit Memory Transfer:* Each Partition allocates extra memory (i.e., halo regions) to store remote metadata and the *haloUpdate* method executes explicit memory transfers between GPUs. The required number of transfers depends on the partitioning schema and the Partition memory layout. Data marshaling may be needed to reduce the required memory transfers.
- *Unified Memory:* Partitions do not allocate halo regions and, instead, rely on the device driver to automatically

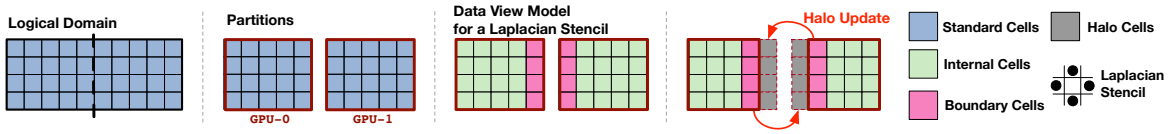


Fig. 3: Data view model where different grid cells are categorized based on their dependencies.

manage the coherency of the memory between different GPUs. While the unified memory system simplifies the implementation, it comes with a performance penalty due to page faults that could only be mitigated by guiding the unified memory driver to eventually achieve the same performance as explicit memory transfer [19].

The dense and element-sparse Grids provided by Neon leverage the explicit transfer strategy since it provides full control over memory management which is essential for OCC optimizations (Section V). In shared-memory multi-GPU systems, the number of GPUs is relatively low, both Grids decompose the Cartesian domain only on one dimension so that each GPU communicates only with two other neighbour GPUs. For scalar Fields, both Grids layouts organize *Boundary* cell metadata in contiguous segments—for each Partition the *haloUpdate* executes only two memory transfers and no marshaling is needed. The same is true for vector Fields with an AoS organization. For n-component vector SoA Fields, *Boundary* cells metadata is organized in contiguous regions for each component—for each Partition the *haloUpdate* executes $2n$ transfers, no marshaling is needed.

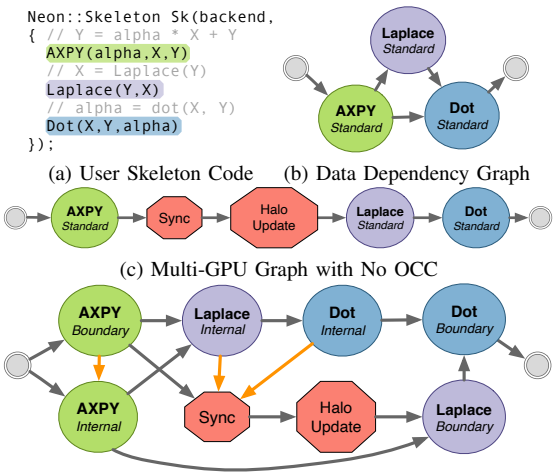
V. The Skeleton Abstraction

The Skeleton abstraction represents the highest abstraction in Neon and is the level at which users will write their code, as presented in Section III. In the previous abstractions, a lot of complexity is still left with a parallel computing background to manage—requesting halo updates, multi-GPU synchronizations, and optimally organizing the execution.

The Skeleton is Neon's orchestrator. Starting from sequential user code (e.g., Listing 3), the Skeleton automatically composes an execution graph, and optimizes it for efficient execution on the target back end. Our primary optimization objectives are to fully leverage the application concurrency and to schedule the entire multi-GPU execution to best overlap computation and communication. Under the hood, the Skeleton abstraction level operates with three main steps with a focus on a multi-GPU back end:

- Extracting a data dependency graph from the user code
- Creating a multi-GPU optimized version of the graph
- Identifying a scheduling strategy for the execution

All three steps are automatically managed within Neon, and no user intervention is required. In the following, we use an example code snippet in Fig. 4a and track it through each of the three stages of the Skeleton. The code snippet is composed of three Containers: a map operation (`axpy`), a user-defined stencil operation (`laplace`), and a reduce operation (`dot`).



(a) User Skeleton Code (b) Data Dependency Graph

(c) Multi-GPU Graph with No OCC

(d) Multi-GPU Graph with Two-way Extended OCC

Fig. 4: From a sequential user code (a), Neon creates a data dependency graph of the application (b) that ensures a correct execution. Then, Neon creates a multi-GPU graph based on the selected optimization e.g., No OCC (c) or Two-way Extended OCC (d). Green blocks are map computations, purple blocks are stencil operations, and blue blocks are reductions. Red blocks are multi-GPU synchronizations e.g., barriers and halo updates. Gray arrows are data dependencies, and orange arrows are scheduling hints.

Collectively, they represent the three building blocks used for application authoring in Neon.

A. Data Dependency Graph

The Skeleton generates a data dependency graph of the user application where nodes are Containers, and edges are dependencies between Containers that read or write the same Multi-GPU data objects. Here we rely on the solution for the Dependency-graph Challenge provided by the Set abstraction (Section IV-B3). Once the dependency graph is fully populated, each node is initialized with a reference to its Container and a set of flags—operation type (`MapOp`, `StencilOp`, `ReduceOp`), data view model (Standard, Internal and Boundary) and, a coherency flag. The coherency flag is used to track Containers that need a halo update for stencil operations.

In our example, the resulting graph (Fig. 4b) shows the generated dependency (gray arrows) between `axpy` and `laplace` Containers on the `X` and `Y` Multi-GPU data with *write-after-read* (WaR) and *read-after-write* (RaW) types, respectively. Similar dependency is generated between the `laplace` and `dot` Containers. `axpy` is marked as `MapOp`,

`laplace` as `StencilOp`, and the `dot` as `ReduceOp`. All nodes in the graph are associated with a Standard data view. Only the `laplace` Container is flagged as *incoherent* since it requires a halo update operation on `X`.

B. Multi-GPU Graph

The next step is to transform the data dependency graph into a *multi-GPU graph* that takes into consideration halo updates, synchronizations, and possible optimizations. Fig. 4c shows the resulting multi-GPU graph with no optimizations applied. Neon adds halo update nodes to ensure the stencil operation nodes operate on the latest halo data values. At last, the dependency between the map and the dot product nodes is removed as redundant for the execution.

Optimizations: While constructing the multi-GPU graph, we support the following optimizations:

- *Standard OCC* is a well-known technique that works by splitting stencil computation into two Containers—one on internal cells, the other on boundary cells. Since only the boundary cells depend on data from neighbouring partitions, we can overlap the halo update communications with the internal cells computation.
- *Extended OCC* where the stencil split is propagated to all map nodes preceding the stencil node. As a result, communication can be overlapped with the internal cells computation of both the map and the stencil operations.
- *Two-way Extended OCC* where the stencil split strategy also includes map or reduce nodes **after** the stencil node (Fig. 4d). When splitting a reduction node (e.g., the dot product), a data dependency is also added between the internal and the boundary cells computations. In our example, the halo update for the stencil operation can potentially be overlapped with the internal map, internal stencil, and internal reduce operations nodes.

The potential overlap between memory transfers and computation becomes a concrete optimization only if the scheduling process detects it and produces an execution plan accordingly. For example, in Fig. 4d, the internal map operations should only be launched after the boundary map operation, otherwise no overlapping is possible. We leverage the knowledge of the computation structure during the optimization of the multi-GPU graph, where Neon adds *scheduling* hints to the graph (orange arrows). While a data dependency forces a node to be completed before the dependent one starts, a scheduling dependency is a hint to launch a node before another one. For example, in the multi-GPU graph in Fig. 4d, we provide two hints to the scheduler to launch the internal stencil operation node and the reduce internal node before the synchronization node to ensure communication/computation overlap.

C. Scheduling a Multi-GPU Graph

The scheduling process computes a plan to execute the multi-GPU graph i.e., how to concurrently launch and synchronize all the kernels on each GPU and to satisfy any data dependency. The scheduling algorithm defines a set of operations for the specific queue-based execution model defined by the System level and extended to multi-GPU environments by the Set level. The Skeleton scheduling algorithm outputs a list of tasks to be processed in the specified order by the host when the graph is executed. Each task is associated with one multi-GPU graph node and it includes the following information:

- A target multi-GPU Stream
- A wait list of multi-GPU Events that must be completed before executing
- A reference to a multi-GPU graph node
- A multi-GPU Event to signal the completion of the task

The execution of a task works by enqueuing asynchronous commands in the task multi-GPU Stream. Firstly, we request the task stream to stop dispatching any operation until other multi-GPU Streams have processed all the multi-GPU Events in the event list. The barrier enforces the data dependencies between concurrent multi-GPU graph nodes. Then, we request the stream to process the multi-GPU graph node operation (computation, memory transfer, or synchronization). Finally, we enqueue the completion Event to signal the completion of the multi-GPU graph node operation asynchronously. For efficiency purposes, some synchronizations are skipped when two dependent nodes are executed on the same stream.

To build our ordered list of tasks, we rely on a greedy approach composed of three phases: mapping nodes to streams, organizing Events to ensure data dependencies, and finally defining the task list order.

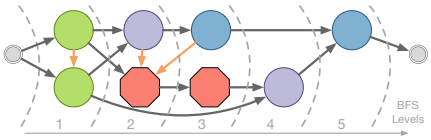


Fig. 5: BFS levels for stream mapping based on Fig. 4d.

a) Mapping Nodes to Streams: We visit the multi-GPU graph (only the data dependency arrows) in a breadth-first search (BFS) fashion while tracking node data dependencies—a node can be added to the BFS frontier only if all its parents have already been processed (Fig. 5). The result is an ordered sequence of *levels*, where each level contains independent nodes, and so they can be run concurrently on different streams. The maximum number of nodes per level determines the number of streams needed to execute the graph. We then process level by level by matching one of the allocated streams to each multi-GPU node. If possible, we give a node the same stream used by one of its parents located in previous levels. This operation reduces Events synchronization overhead.

b) Organizing Event Synchronization: In a second pass of the graph, we add the needed synchronizations through Events. Given a node and its parents, we first compare their streams. If the streams do not match, synchronizations are added by allocating a completion Event for the parent task and inserting it into the child task event wait list.

c) Task List Order: We perform a new BFS traversal on the graph containing both data dependencies and scheduling hints (Fig. 6). We finally order the tasks starting from the first

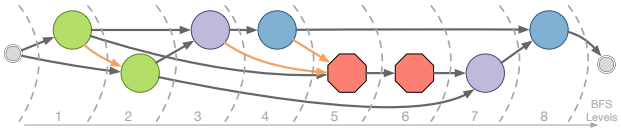


Fig. 6: BFS levels for task ordering based on Fig. 4d.

level to the last—tasks at a level should come before any task in the following level. We do not impose any specific order for tasks in the same level.

Size	Neon (LUPS)	Taichi (LUPS)	Speedup
4096×1024	33346.73	29304.42	1.14
8192×2048	33740.24	33979.88	0.99
16384×4096	34001.84	34729.78	0.98
32768×8192	206079.33	206127	0.999

TABLE I: Neon's speedup over Taichi on the 2D Kármán Vortex Street LBM application using LUPS metric. Experiments were run on a single GPU of a DGX A100 server.

D. Compiler Approach v.s. Library Approach

To build a multi-GPU programming framework, a compiler-based approach has more powerful code analysis and generation capabilities that are beneficial for single-GPU optimizations. We argue that a library approach did not impact Neon's capabilities for multi-GPU-specific optimizations (i.e., OCC) since the execution of the multi-GPU computation graph is a runtime operation. The only limitation that this design decision incurs is the inability to optimize the single-GPU performance (e.g., via kernel/container fusion and tiling). We leave exploring integrating Neon as a runtime for a compiler to future work. In order to provide a better insight towards Neon single GPU performance against a compiler-based alternative, we compare against Taichi [13] using 2D Kármán Vortex Street LBM application (Table I) where the two systems' performance closely match.

VI. Results

We selected three real-world applications to test Neon's performance, each also highlighting different aspects of Neon. A Lattice-Boltzmann Method (LBM) application compares Neon's performance against both a well-known CUDA benchmark and an implementation that leverages a high level parallel abstraction recently introduced in C++. A finite-difference Poisson solver shows Neon closely matching the performance of a plain CUDA implementation. It also showcases the impact of different OCC optimizations. Finally, a finite-element linear-elastic solver explores the trade-off between dense and sparse data structures—a user controlled parameter. We use strong scalability as the primary performance metric for all applications. Scalability results are presented according to parallel efficiency defined as: $\text{Efficiency}(n) = t_{baseline}/(n\ t_n)$, where n is the number of GPUs used, t_n is the time for a benchmark running on n GPUs, and $t_{baseline}$ is the baseline implementation running on a single GPU.

Size	cuboltz	stlbm AA	stlbm twoPop	Neon twoPop
128^3	4045	3553 (-12%)	3085(-23%)	4019 (-0.6%)
192^3	4268	3689 (-13%)	3488(-18%)	4114 (-0.3%)
256^3	4356	3708 (-14%)	3636(-16%)	4206 (-0.3%)
320^3	4390	3705 (-15%)	3646(-16%)	4224 (-0.3%)

TABLE II: LBM performance reported in MLUPS, followed in parenthesis by the gain with respect to cuboltz. Experiments were run on a single GPU of a DGX A100 server.

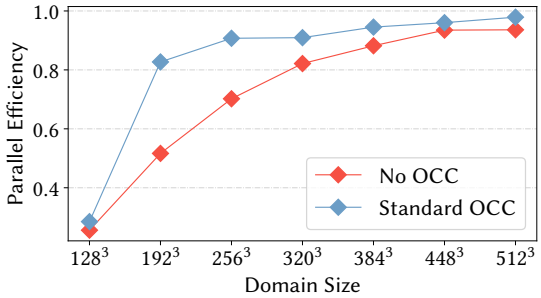


Fig. 7: Neon's parallel efficiency of twoPop LBM using 8 GPUs on DGX A100 server. As reference for the efficiency, Neon twoPop on single GPU is used.

We conducted our experiments on two systems. The first one is an Nvidia DGX A100 with 8 GPUs (40GB HBM2e) and NVLink fast interconnect, running DGX OS 5 (Nvidia docker image cuda:11.4.1-devel-ubuntu20.04). The second system is a two-socket Intel Xeon (E5-2640) with 8 Nvidia Quadro GV100 GPUs (32GB HBM2) connected to a Gen 3 PCIe, running Centos 7 (GCC 8.3 and CUDA 11.3).

A. Lattice-Boltzmann Fluid Solver

LBM is composed of two main operations: collide and streaming. The former works on cell local data, the latter computes a stencil operation over the 19 point lattice structure (D3Q19). Both collide and streaming work on input and output fields with 19 components per grid cell. Collide and streaming steps are fused into a single kernel to reduce memory transfers. In the project stlbm, Latte [20] analyzes the performance of an LBM lid-driven-cavity flow benchmark on C++17 parallel algorithms (CPA). A strong contribution of stlbm is testing CPA performance portability. CPA allows code to be executed on both CPUs and GPUs since both Nvidia and AMD provide CPA-enabled compilers for their GPUs. The authors also compared against cuboltz—a CUDA native LBM benchmark.

Stlbm has three variants of LBM (AA, twoPop, Swap) and we implemented the twoPop variant in Neon with some minor changes. The selected variant works on two buffers, one used as input the other as output. At the end of each iteration the two buffers are swapped.

Table II reports the number of million lattice updates per seconds (MLUPS) achieved by the different implementations

running on a single Nvidia A100 GPU. Neon's twoPop performs close to the cuboltz CUDA benchmark, with less than 1% performance degradation. Neon's twoPop outperforms both stlbm's AA and twoPop variants and it can run on multi-GPU systems without changing the user code.

Since twoPop LBM is composed of a single kernel, only Standard OCC could be applied. We analyzed the performance on the Nvidia DGX A100 machine for different domain sizes. In Fig. 7, we report Neon twoPop parallel efficiency running on 8 GPUs using the single GPU implementation as the baseline. Standard OCC yields better parallel efficiency over all domain sizes. Thanks to the fast GPU interconnect, the No OCC version reaches 93% efficiency with 8 GPUs on the biggest domain. The recorded trends are explained by the fact that the bigger the domain, the lower the impact of the communication overhead—with No OCC, 8 GPUs and a 192^3 domain, the communication time is around 49% of the iteration. With a 512^3 domain, this drops down to 10%, therefore, communication has a lower impact on efficiency.

B. Finite-difference Poisson Solver

We implemented the Poisson solver using a standard finite-difference discretization [21] on a Cartesian grid. We used a standard 7-point stencil (Listing 2) to approximate the Laplacian operator and a matrix-free conjugate gradient solver (Listing 3). In order to enable the Two-way Extended OCC, we moved the last map operation (`UpdateP`) to be at the start of the Container followed by the stencil operation (`LaplacianStencil`). Such a change does not impact the correctness of the computation.

We implemented the same solver using CUDA and cuBLAS and considered it as a baseline ($t_{baseline}$). We used cuBLAS to implement the dot product and DAXPY to obtain best-in-class performance for these operations. Our implementation of the stencil for the baseline matches that we have implemented in Neon. Our baseline implementation achieves more than 95% of the GPU's peak effective bandwidth. Fig. 8 (top) shows that Neon incurs a minimal overhead compared to the hardwired application-specific implementation. This overhead is mainly due to Neon's checks to prevent out-of-bound accesses when applying the stencil operations which were not necessary for the baseline implementation. With increased computation, this overhead decreased even further. Given enough parallelism, Fig. 8 (bottom) shows that our different OCC optimizations are effective and can reach ideal efficiency.

The impact of different OCC optimizations are shown in Fig. 8 (top) that emphasizes the importance of the different OCC configurations. We observe that there is no single OCC optimization that always outperforms the others—Standard OCC is the best with 4 GPUs or less, Extended OCC is better for 5 GPUs, and Two-way Extended OCC excels with 6 GPUs or more. This emphasizes the importance of having a system where switching between different optimizations is as simple as changing a single parameter. Neon allows the user to explore different possibilities depending on the problem and system configurations.

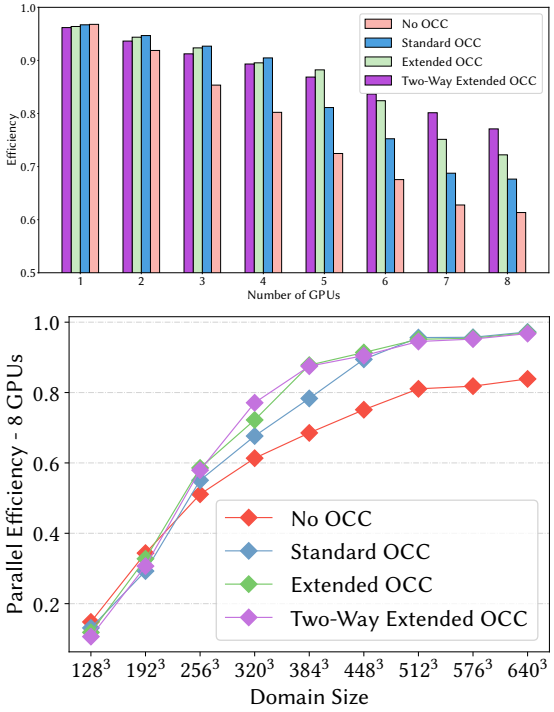


Fig. 8: Poisson solver: impact of different OCC configurations on a 320^3 grid with increasing number of GPUs (top). Parallel efficiency on different grid sizes on 8 GPUs (bottom).

C. Finite-element Linear Elastic Solver

We implemented a matrix-free finite-element linear elastic solver [22] using a 27 point stencil based on the conjugate gradient method (Listing 3). The selected benchmark is based on a solid cube domain with Dirichlet boundary conditions fixing displacements to 0 in the $z = 0$ plane, and Neumann boundary conditions on the rest of the boundary such that the $z = N - 1$ plane had outward pressure applied.

Thanks to the ease of changing the data structures without changing the computation code, we were able to compare the performance of the dense and sparse gird on different grid sizes and sparsity ratios as shown in Fig. 9. The ratio of the solid cube's size to the grid size was tuned to obtain different sparsity levels—fully dense (1.0) and sparse (0.2). The benefits of the element-sparse data structure was clear once the sparsity ratio dropped below 0.8. On the other hand, the dense grid performed better and used less memory when the domain was fully dense as evident on grid size 512^3 and sparsity ratio of 1, where the sparse data structure ran out of memory. This illustrates how the modular structure of Neon can facilitate exploration and optimization of different data representations for user applications.

VII. Conclusion, Limitations, and Future Work

We presented Neon, a programming model and C++ framework for grid-based multi-GPU computation. The system allows the user to compose their applications in terms of a sequence of building blocks. Neon parses these building

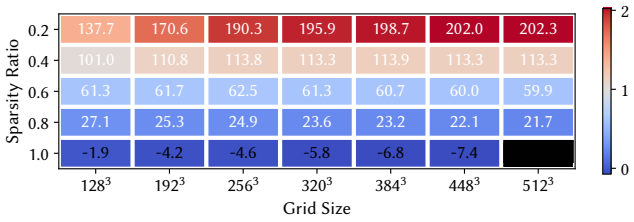


Fig. 9: The ratio (on log scale) between dense and sparse grid data structure running time on the FEM solver using different sparsity ratios and grid sizes. Negative values mean dense grids performs better.

blocks, builds a dependency graph, and injects the necessary optimizations for best scalability. We have shown the importance of these automatic optimizations on real-world applications. Thanks to Neon's hierarchical abstraction, it is possible to switch between different data structures and back ends allowing for rapid prototyping and experimentation. We have shown that the framework can match the performance of a well-optimized CUDA implementation.

We highlight some limitations of the presented approach:

- Our data structures only support local operations, however some applications, like ray-tracing, require global accesses.
- We do not support automatic Container fusion, instead we rely on the user's implementation. This is a restriction that comes from a library-based approach.

There are several avenues for future work:

- We plan to support multi-resolution grid to allow even higher precision where required.
- We wish to implement more multi-GPU graph optimizations, particularly loop unrolling which would extend the applicability of OCC optimization and reduce the synchronization between iterations.
- Distributed systems are a natural extension for Neon. We are evaluating previous research on the concept of cluster-as-accelerator [23] to fully hide the MPI SPMD programming model from the users.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," Google Inc., Tech. Rep., nov 2015, http://download.tensorflow.org/paper/whitepaper2015.pdf.

[2] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 13, Jun. 2013, pp. 519–530.

[3] Apache Software Foundation, *Hadoop*. [Online]. Available: https://hadoop.apache.org

[4] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, Á. L. García, I. Heredia, P. Malík, and L. Hluchỳ, "Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey," *Artificial Intelligence Review*, vol. 52, no. 1, pp. 77–124, 2019.

[5] H. Kuchen, "A skeleton library," in *Euro-Par 2002: Proceedings of the 8th International European Conference on Parallel and Distributed Computing*, ser. Lecture Notes in Computer Science, B. Monien and R. Feldmann, Eds., vol. 2400. Berlin, Heidelberg: Springer, 2002, pp. 620–629.

[6] A. Ernstsson, L. Li, and C. Kessler, "Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems," in *International Journal of Parallel Programming*, vol. 46, Feb. 2018, pp. 62–80.

[7] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[8] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: Overview and status," September 2014. [Online]. Available: https://www.osti.gov/biblio/1169830

[9] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W. E. Nagel, and M. Bussmann, "Alpaka - an abstraction library for parallel kernel acceleration." IEEE Computer Society, May 2016. [Online]. Available: http://arxiv.org/abs/1602.08477

[10] J. Cercos-Pita, "AQUAgpusph, a new free 3d SPH solver accelerated with OpenCL," *Computer Physics Communications*, vol. 192, pp. 295–312, 2015.

[11] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde, "WaLBerla: HPC software design for computational engineering simulations," *Journal of Computational Science*, vol. 2, no. 2, pp. 105–112, 2011.

[12] "Advanced simulation library," 2018. [Online]. Available: http://asl.org.il

[13] Y. Hu, T.-M. Li, L. Anderson, J. Ragan-Kelley, and F. Durand, "Taichi: a language for high-performance computation on spatially sparse data structures," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 6, p. 201, 2019.

[14] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith, "The OPS domain specific abstraction for multi-block structured grid computations," in *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, New Orleans, LA, USA, 2014, pp. 58–67.

[15] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[16] X. Guo, M. Lange, G. Gorman, L. Mitchell, and M. Weiland, "Developing a scalable hybrid MPI/OpenMP unstructured finite element model," *Computers & Fluids*, vol. 110, pp. 227–234, 2015, ParCFD 2013.

[17] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," Aug. 1994. [Online]. Available: https://www.cs.cmu.edu/ quake-papers/painless-conjugate-gradient.pdf

[18] V. Amaral, B. Norberto, M. Goulão, M. Aldinucci, S. Benkner, A. Bracciali, P. Carreira, E. Celms, L. Correia, C. Grelck, H. Karatza, C. Kessler, P. Kilpatrick, H. Martiniano, I. Mavridis, S. Pllana, A. Respício, J. Simão, L. Veiga, and A. Visa, "Programming languages for data-intensive HPC applications: A systematic mapping study," *Parallel Computing*, vol. 91, p. 102584, Mar. 2020.

[19] NVIDIA Corporation, "CUDA C++ programming guide," Nov. 2021, pG-02829-001_v11.5. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[20] J. Latt, C. Coreixas, and J. Beny, "Cross-platform programming model for many-core lattice Boltzmann simulations," *PLOS ONE*, vol. 16, no. 4, pp. 1–29, apr 2021.

[21] S. C. Chapra and R. Canale, *Numerical Methods for Engineers*, 5th ed. USA: McGraw-Hill, Inc., 2005.

[22] A. F. Bower, *Applied Mechanics of Solids*. CRC Press, 2009.

[23] M. Drocco, C. Misale, and M. Aldinucci, "A cluster-as-accelerator approach for SPMD-free data parallelism," in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE, 2016, pp. 350–353.