Lightning: Scaling the GPU Programming Model Beyond a Single GPU

Heldens, S.; Hijma, P.; van Werkhoven, B.; Maassen, J.; van Nieuwpoort, R.V.

DOI
10.1109/IPDPS53621.2022.00054

Publication date
2022

Document Version
Final published version

Published in
Proceedings, 2022 IEEE 36th International Parallel and Distributed Processing Symposium

Lightning: Scaling the GPU Programming Model Beyond a Single GPU

Stijn Heldens*†, Pieter Hijma†‡, Ben van Werkhoven*, Jason Maassen*, Rob V. van Nieuwpoort*†
*Netherlands eScience Center, †University of Amsterdam, ‡Vrije Universiteit Amsterdam
{s.heldens, b.vanwerkhoven, j.maassen, r.vannieuwpoort}@esciencecenter.nl, pieter@cs.vu.nl

2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-8106-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPS53621.2022.00054

***Abstract*—The GPU programming model is primarily aimed at the development of applications that run one GPU. However, this limits the scalability of GPU code to the capabilities of a single GPU in terms of compute power and memory capacity. To scale GPU applications further, a great engineering effort is typically required: work and data must be divided over multiple GPUs by hand, possibly in multiple nodes, and data must be manually spilled from GPU memory to higher-level memories.**

We present *Lightning*: a framework that follows the common GPU programming paradigm but enables scaling to large problems with ease. Lightning supports multi-GPU execution of GPU kernels, even across multiple nodes, and seamlessly spills data to higher-level memories (main memory and disk). Existing CUDA kernels can easily be adapted for use in Lightning, with data access annotations on these kernels allowing Lightning to infer their data requirements and the dependencies between subsequent kernel launches. Lightning efficiently distributes the work/data across GPUs and maximizes efficiency by overlapping scheduling, data movement, and kernel execution when possible.

We present the design and implementation of Lightning, as well as experimental results on up to 32 GPUs for eight benchmarks and one real-world application. Evaluation shows excellent performance and scalability, such as a speedup of 57.2× over the CPU using Lighting with 16 GPUs over 4 nodes and 80 GB of data, far beyond the memory capacity of one GPU.

***Index Terms*—GPU, distributed computing, CUDA, programming model**

I. Introduction

Many applications in industry/science are nowadays accelerated by *Graphics Processing Units* (GPUs) [1]–[4] and GPUs will likely be used in future exascale systems [5]. A GPU application consists of GPU-specific functions (called *kernels*) that are executed on the GPU by a large number of threads in parallel. This massive parallelism provides excellent speedups over the CPU, but a single GPU is limited for large problems that exceed the GPU capabilities

There are three orthogonal solutions to increase scalability: 1) spill data from GPU memory to host memory (or even disk), 2) use multiple GPUs within one node, or 3) use a cluster of GPU-accelerated nodes. For all these solutions, the programmer must manually split the data into smaller pieces and either stream these pieces through GPU memory, when using a single GPU, and/or distribute them among different memories, when using multiple GPUs. Data must be communicated between GPUs to maintain data consistency and this intra- and inter-node communication should be overlapped with kernel execution to avoid idle time [6]. Each kernel launch must also be split into smaller launches and scheduled onto the available GPUs while maintaining correctness. Additionally, GPU kernel code must be heavily modified to change indexing into data structures and account for offsets in thread indices. Finally, different tools and libraries must be combined (e.g., MPI, threading, serialization, scheduling, etc.). All of this together is a massive engineering effort that leads to complex code that is difficult to develop and maintain [7].

Several frameworks have been proposed to aid the development of distributed multi-GPU applications either by facilitating local access to remote GPUs for CUDA [8]–[13] or OpenCL [12]–[18], by abstracting multiple (remote) GPUs into a single virtual device [19]–[21], or by offering special distributed data structures [22]–[24]. However, no framework alleviate the programmer of all of the above complexities.

In this work, we present *Lightning*: a framework that enables programmers to use a GPU-accelerated cluster in a way that is similar to programming a single GPU, without worrying about low-level details such as network communication, memory capacity, and data transfers. Lightning supports *distributed kernel launches*, which enable multi-GPU execution of a single kernel, and *distributed arrays*, which distribute data using a user-specified policy. Existing CUDA kernels can be used in Lightning with only minor modifications. Data access annotations on kernels allow Lightning's runtime system to automatically infer their data requirements, as well as the data dependencies between subsequent kernel launches. This enable multi-kernel workflows and complex pipelines.

All in all, Lighting provides many features that alleviate programmers from the concerns of multi-GPU programming:

- Support for *distributed kernel launches* that automatically distribute the work for a single kernel launch across the available GPUs in a cluster.
- Existing CUDA kernel code can be reused by making only slight changes and providing data annotations.
- Support for multi-dimensional *distributed arrays* that have their data transparently distributed across the cluster.
- Data is automatically spilled to higher-level memory, enabling datasets that do not fit into GPU memory.
- Data can be (partially) replicated among multiple GPUs and replications are automatically kept consistent.
- Focus on asynchronous processing to enable overlapping of scheduling, data movement, and kernel execution.
- Data dependencies between consecutive kernel launches are automatically detected and tasks are executed in parallel in a sequentially consistent order [25].

1530-2075/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPS53621.2022.00054

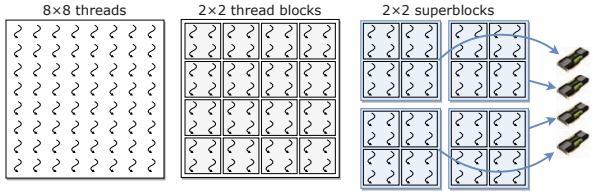


Fig. 1: Example superblock distribution for 8×8 grid.

In this paper, we present the design and implementation of our framework, as well as experimental results for eight benchmarks and one real-world application. Evaluation shows many excellent results, such as $57.2\times$ speedup over the CPU using Lightning with 16 GPUs for a dataset of 80 GB, which is far beyond the memory capacity of a single GPU.

II. Design

In this section, we present the abstractions that Lighting offers to distribute work (*distributed kernel launches*, Sec. II-A) and distribute data (*distributed arrays*, Sec. II-B). These two concepts are united by *data annotations* (Sec. II-C) that allow the *planner* (Sec. II-D) to construct an execution plan.

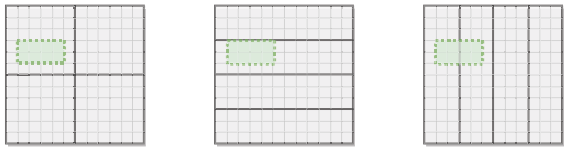
A. Distributed Kernel Launches

In GPU programming (e.g., CUDA or OpenCL), work is performed on the GPU by launching a *kernel* onto the device. Kernels are GPU-specific functions that are executed by a large number of GPU threads in parallel. A kernel launch is initiates an n-d grid of threads ($n = 1, 2, 3$) where each thread is assigned a unique n-d index. Additionally, the threads are grouped into fixed-sized rectangular *thread blocks*. Threads within the same thread block can communicate, while threads from different thread blocks cannot synchronize and run independently of each other[1].

For Lightning, we exploit the fact that thread blocks are independent by distributing the thread blocks of a single kernel launch across multiple GPUs, thus enabling multi-GPU execution of a single kernel. We call this a *distributed kernel launch*. The distribution of work is achieved by grouping thread blocks into rectangular disjoint subgrids that we call *superblocks*. Fig. 1 shows an example. Each superblock is essentially one job: each is assigned to one GPU in the system and that subset of thread blocks will be executed on that specific GPU. The superblock distribution must be passed explicitly by the programmer for each kernel launch.

Currently, Lightning supports kernels written in CUDA, although we plan on also supporting other kernel languages. Small modifications need to be made to the kernel code to make existing CUDA kernels compatible with our framework, such as using Lightning-specific data types (see Sec. III-F)

[1]Recent versions of CUDA added *cooperative* kernels where synchronization across thread blocks is possible, but we focus on conventional kernels.



(a) Tile distribution. (b) Row-wise dist. (c) Column-wise dist.

Fig. 2: A 12×12 array partitioned according to three distributions. The black rectangles indicates chunks. The dashed rectangle is an example of the access region of a superblock.

B. Multi-Dimensional Distributed Arrays

Besides distributing work, it is also necessary to distribute data. GPU applications typically use multi-dimensional arrays (e.g., vectors, matrices, tensors) as their predominant data structures since they fit the data-parallel model of GPUs. Therefore, Lighting supports multi-dimensional arrays as its primary data abstraction. These arrays can be created/deleted dynamically at runtime, have up to three dimensions, and store elements of a primitive type (e.g., `int`, `float`).

Similar to how the threads of a kernel launch must be distributed across GPUs, the data elements of an array also needs to be distributed. In Lighting, the programmer has to specify the distribution policy for each array. Such a policy defines a set of rectangular subregions called *chunks* that together cover the entire domain of the array (see Fig. 2). Each chunk is assigned to one GPU in the system. Several common distributions are included in Lightning (e.g., row/column-wise, tiled) and custom distributions can also be defined.

Whereas superblocks must be disjoint (i.e., each thread is assigned to exactly one superblock), the chunks of one data distribution may overlap (i.e., one data element can be assigned to multiple chunks). This is useful, for example, for stencil distributions that add a border of halo cells around each tile. The replicated data elements are automatically kept coherent by Lightning's runtime system.

Although each chunk is assigned to one specific GPU, Lightning will automatically spill the chunk's content from GPU memory to higher-level memories if GPU memory is full (See Sec. III-D). It is thus recommended to create chunks with a limited size, allowing the runtime system to overlap kernel execution with transferring chunks into and out of GPU memory. We found chunks around $\sim$0.5GB to give good performance (see Sec. IV-C).

C. Data Annotations

For each distributed kernel launch, the programmer must specify the arrays that will be accessed by the launched threads. To be able to distribute these threads across multiple GPUs, we need some way to determine what parts of these arrays are accessed by the threads. Typically in GPU programming, each thread accesses only a few elements, but this information is normally not encoded into the kernel code.

For Lightning, we define the *access region* of a superblock for an array as the n-d dense rectangular area (i.e., lower and

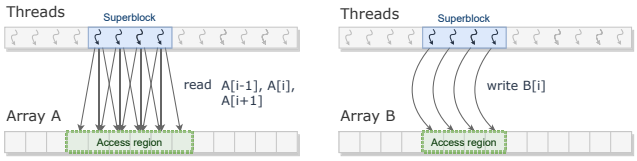


Fig. 3: Example of superblock and associated access regions.

upper bounds along each axis) that will be accessed by the threads in that superblock. As an example, consider a simple 1D stencil kernel where thread i performs $B[i] = A[i-1] + A[i] + A[i+1]$. Fig. 3 demonstrates the access regions on A and B for one superblock.

To specify these access regions per superblock, Lightning offers a symbolic notation to describe the access pattern of each thread. By annotating kernels, Lightning can automatically extract the access regions for each superblock. For example, we can formalize the above stencil access pattern by stating that thread i `reads` elements $A[i-1], \ldots, A[i+1]$ and `writes` element $B[i]$. The data annotation in Lightning for this example is as follows:

```
global i => read A[i-1:i+1], write B[i]
```

This annotation should be interpreted as follows. To the left side of the arrow are variable bindings that, in this case, bind the `global` 1D thread index to variable i. Other possible bindings are `block` (thread block index) and `local` (local index within block). To the right side of the arrow are statements that describe, for each argument array, the *indices* that are accessed and the *access mode*. Each index can either be a single expression or a Fortran-style slice notation "*lower bound* : *upper bound*" (both bounds can be omitted). Each index expression must be a linear combination of the bound variables to simplify analysis of the access pattern.

For the access mode, there are four supported options:

- `read`: Access is read-only. Writes are not permitted.
- `write`: Access is write-only. Reads are not permitted.
- `readwrite`: Access is both `read` and `write`.
- `reduce(f)`: Similar to `write`, except 'conflicting' writes are reduced (`f` must be `+`, `*`, `min`, or `max`).

Another example of an annotation is for a naive matrix multiplication kernel performing $C = AB$ where thread (i, j) writes entry C_{ij}, reads row i of A, and reads column j of B.

```
global [i, j] => read A[i,:], read B[:,j],
                 write C[i,j]
```

Yet another example is a reduction of matrix A along the columns to a vector `sum`. Thread (i, j) reads A_{ij}, threads cooperatively reduce values and write their results to $\texttt{sum}_i$.

```
global [i, j] => read A[i,j], reduce(+) sum[i]
```

For the reductions, Lightning internally allocates temporary memory to which the threads can write their local results. Afterward, Lightning performs a multi-level reduction.

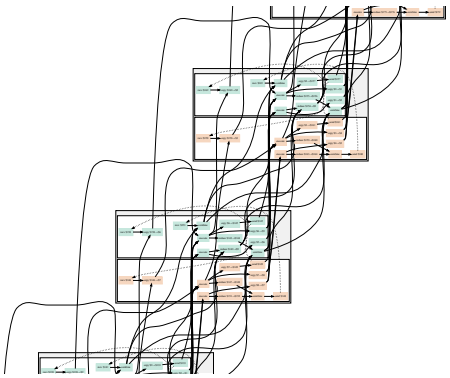


Fig. 4: DAG created for stencil kernel (Fig. 9). Shows four iterations on two nodes with two GPUs per node. Large boxes represent distributed kernel launches and smaller colored boxes represent individual tasks (color indicates the node).

D. Execution Planner

For each distributed kernel launch, Lightning will construct an *execution plan*. Such a plan consists of a *directed acyclic graph* (DAG) for each node in the system containing the tasks for that node and the dependencies between tasks. Examples of DAG tasks are `Execute` a kernel, `Create/Delete` a chunk, `Copy` data between chunks and `Send`/`Recv` chunks between nodes. Fig. 4 shows an example of an execution plan.

Execution plan construction is performed by the *planner*. First, the planner divides the kernel launch into superblocks. For each superblock, the planner processes each argument array. For each argument, the planner first evaluates the data annotation to determine the access region and then queries the array's data distribution to determine which chunks intersect the access region. In the common case, data is distributed such that there will be at least one chunk enclosing this access region (see Figs. 2a and 2b). If that chunk is assigned to the superblock's GPU, then the chunk can be used directly. Otherwise, it must be copied between GPUs, or even between nodes, by inserting `Copy`/`Send`/`Recv` tasks into the DAG. For `write` accesses, the planner also inserts proper data transfers to update replicated data elements.

In exceptional cases, the access region might intersect with multiple chunks (Fig. 2c). For `read` access, the planner assembles a temporary chunk from the contents of the intersected chunks. For `write` access, the planner creates a temporary uninitialized chunk and afterward scatters its content. While this procedure might be inefficient, it means data distributions only affect the *performance* of an application and not the *correctness*. This provides separation of concerns: programmers can first develop the application and later tune the work/data distributions to maximize performance.

The planner handles `reduce` accesses separately. For each superblock, a temporary chunk is created to hold the block-level partial results. Afterward, the planner inserts reduction tasks to hierarchically reduce the partial results: first the results for one superblock, then for one GPU, then for each node, and finally reducing the results across all nodes.

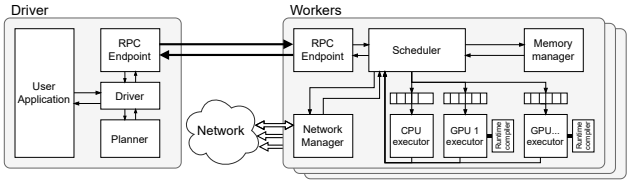


Fig. 5: Overview of Lightning's runtime system.

After the execution plan for one distributed kernel launch has been constructed, the DAGs are immediately submitted to the nodes in the system. Execution on the host continues, allowing additional kernels to be launched. This increases efficiency since it overlaps plan construction with kernel execution and data movement on the nodes. However, this means that tasks from the previously submitted DAGs might not have finished when the next kernel is already being planned. To solve this problem, the planner analyzes the dependencies between consecutive distributed kernel launches and inserts dependencies from previously submitted tasks when there are data conflicts on chunks (i.e., read-write/write-write/write-read conflicts). Essentially, the planner incrementally builds a large DAG from many smaller DAGs.

E. Scope and Limitations

While the design of Lightning is versatile, there are limitations. First, while the data annotations are simple and expressive, they require the data access pattern to be predictable and derivable from the thread/block indices. This is the case for regular algorithms, for example those from linear algebra. However, data-dependent problems generally cannot be expressed. In some cases, they can be described imprecisely, resulting in a performance penalty. For example, sparse matrix-vector multiplication (SpMV) performs unstructured reads on the input vector, but can be still be expressed by overestimating the access region to be the entire vector (see Sec. IV-B).

Second, Lightning supports multi-dimensional arrays since they fit well into the data-parallel model of GPUs. Irregular data structures, such as linked lists or graphs, are unsupported. Additionally, the data access patterns on the arrays must be dense and rectangular, other patterns cannot be expressed for now (e.g., triangular, diagonals, indirection).

Third, users must manually annotate their code. Automatic extract of data annotations using static code analysis, while interesting, is out of scope for this manuscript.

III. Implementation

In this section, we discuss the implementation of Lightning.

A. Overview

Fig. 5 shows the software architecture of Lightning. Our system runs on a cluster of *worker nodes* which are managed by one central *driver program*. The driver acts as the centralized component in the system: it coordinates the workers, maintains bookkeeping of the distributed arrays, and builds the execution plans. Each worker node is equipped with one or more GPUs and the workers execute the commands submitted by the driver. In our implementation, the driver program also runs on the first worker node, meaning there is no network overhead when using just a single node.

The driver also runs the user's application and each call made by the application into the system is handled by the driver. For instance, when the application creates an array, the driver maintains the associated metadata and requests the workers to allocate the chunks in memory. When the application launches a kernel, the driver builds the execution plan and submits the resulting DAGs to the workers. The rationale behind this choice is that it matches the conventional model of GPU programming where a central host offloads compute-intensive tasks to a discrete GPU.

B. Communication & Data Movement

Lightning uses MPI for the network layer. We use a simple RPC protocol on top of MPI for the communication between the driver and the workers, since this traffic consists solely of small control messages. For communication between the workers themselves, we use non-blocking MPI point-to-point primitives, since this network traffic consists of bulk data exchanges. We assume workers are connected through a fast interconnect (e.g., InfiniBand), although any MPI implementation is compatible with Lightning.

Data transfers between host memory and GPU memory are performed using asynchronous memory copies to allow overlapping data movement with kernel execution. Data transfers between two GPUs on the same node are performed using asynchronous peer-to-peer copies, which uses DMA (*direct memory access*) to directly copy between GPUs. For transfers between GPUs on different nodes, data is staged in host memory and transferred using MPI. This gives sufficient performance since Lighting effectively exploits asynchronous processing to overlap data movement with kernel execution.

C. Scheduling

For each distributed kernel launch submitted by the application, the driver requests the execution planner to construct an execution plan, consisting of a DAG of tasks for each worker. The driver submits these DAGs to the workers and each worker has its own *scheduler* to schedules these tasks onto the local resources. The actual *scheduling* of the DAG is thus done by the workers themselves, while the driver only *plans* the DAG. This is important since DAG tasks can be small (in the order of milliseconds) and centralized scheduling would quickly become a bottleneck.

Initially, each task must wait until its predecessor tasks finish. Once a task's dependencies (i.e., predecessor tasks) complete, the task is ready to be executed. First, the scheduler submits the task to the *memory manager* for *staging*. Each task is associated with several chunks it will access and staging entails that these chunks must be materialized in the requested memory space (see Sec. III-D). Next, after staging completes, the *scheduler* queues the task at the appropriate *executor* (i.e., CPU, GPU, or network). Finally, once the task

```
__global__ void stencil(

        int n,
        float *output,
        const float *input
) {
  int i = blockDim.x  * blockIdx.x + threadIdx.x;
  if (i >= n) return;

  float left = i-1 >= 0 ? input[i-1] : 0;
  float mid = input[i];
  float right = i+1 < n ? input[i+1] : 0;
  float new_val = (left + mid + right) / 3.0;

  output[i] = new_val;
}
```

Fig. 6: Original CUDA source code.

```
__device__ void stencil(
        dim3 virtBlockIdx,
        int n,
        lightning::Vector<float> output,
        const lightning::Vector<float> input
) {
  int i = blockDim.x  * virtBlockIdx.x + threadIdx.x;
  if (i >= n) return;

  float left = i-1 >= 0 ? input[i-1] : 0;
  float mid = input[i];
  float right = i+1 < n ? input[i+1] : 0;
  float new_val = (left + mid + right) / 3.0;

  output[i] = new_val;
}
```

Fig. 7: Modified code from Fig. 6 (Changes in red).

finishes execution, the scheduler requests the memory manager to *unstage* the tasks (i.e., release the task's chunks) and checks which successor tasks are not ready for execution.

When multiple tasks become ready simultaneously, the scheduler selects one arbitrary task without further considerations. We found that this performs adequately in practice since Lightning effectively exploits asynchronous processing. For future work, we will explore more complex scheduling policies that consider, for example, data locality or task priority.

D. Memory Management

Every worker has its own *memory manager* that maintains the bookkeeping of all local chunks and where they are allocated. Each chunk can be allocated in host memory, GPU memory, or disk. The memory manager automatically moves chunks between these different memory spaces when required.

For each task that gets staged, the memory manager's responsibility is to materialize the chunks associated with the task. First, memory must be allocated for chunks that are currently not allocated in the requested memory space. The memory manager uses pre-allocated memory pools because we found allocations of device memory and page-locked host memory to be expensive. It is important that all the task's chunks are allocated in one action to prevent deadlocks. If memory is full, previously allocated unused chunks are evicted in least-recently used fashion to higher-level memory (i.e., GPU to RAM, RAM to disk).

Second, if a the data in the allocated chunk was previously evicted, data must be copied back from the higher-level memory. All data transfers performed by the memory manager are asynchronous. It is important that a sufficient number of tasks is being staged concurrently to enable overlapping work performed by the executors with the staging of future work by the memory manager.

The scheduler must throttle the number of tasks that are staged simultaneously at any moment in time since this number presents a trade-off. On the one hand, allowing too few concurrently staged tasks prohibits overlapping data transfers with task execution. However, on the other hand, allowing too many leads to contention where tasks are staged too far ahead of time. Our current implementation uses a simple heuristic to throttle the number of concurrently staged tasks: the total memory footprint of tasks that are staged onto one resource simultaneously cannot exceed some predefined threshold. We found a threshold of 2 GB to work well in practice.

E. Runtime Kernel Compilation

Lightning supports existing GPU kernels written in CUDA, with minor modifications. To illustrate these changes that one must make, we use an example of a simple stencil operation (see Fig. 6). Three changes must be made by the user (see Fig. 7) before this kernel can be used within Lightning:

- Change the declaration from `__global__` (kernel function) to `__device__` (device function).
- Explicitly take the block index as a parameter. This is required since this index will be virtualized, so the physical block index (`blockIdx` in CUDA) is incorrect.
- Change arguments from raw data pointers to Lightning-specific data types (`Scalar`, `Vector`, `Matrix`, `Tensor` for $0, 1, 2$, or 3-D arrays). These types overload several operators and can be accessed like regular arrays without changing their indexing.

The system performs runtime compilation, meaning that the source code of a CUDA kernel must be provided at runtime; each worker in the system compiles *a local version* of the code and loads the resulting kernel into the GPU at runtime. The regular NVIDIA CUDA compiler is used at runtime for compilation. The advantage of runtime over ahead-of-time compilation is that any runtime constant (for index calculations) can be inserted into the kernel code at compile-time to minimize the overhead of our framework. In Sec. IV-F, we show that runtime compilation means the overhead of Lightning over directly using CUDA is small.

The user's kernel is not called directly, but instead, Lightning generates a wrapper kernel that performs some steps before calling the user's kernel. First, an offset is added to the physical block index and the user's kernel is called with this virtual block index as its first argument. This solves the problem that CUDA always numbers thread blocks from zero. Second, the wrapper is passed chunks that correspond to subregions of larger arrays and, to give the illusion that

```
1  extern "C" __global__ void stencil_wrapper_ftpyotpf8VofcBIdGGEfxrlOdmfpzbWY(
2    int32_t n,
3    float *const output_ptr,
4    const float *const input_ptr
5  ) {
6    // Worker-specific constants
7    const uint32_t block_offset_x = 1024, block_offset_y = 0, block_offset_z = 0;
8    const size_t input_offset_0 = 1023, input_strides_0 = 1;
9    const size_t output_offset_0 = 1024, output_strides_0 = 1;
10
11   // Prepare arguments
12   dim3 virtual_block_index(block_offset_x + blockIdx.x, block_offset_y + blockIdx.y, block_offset_z + blockIdx.z);
13   ::lightning::Array<float, 1> output(output_ptr - output_offset_0 * output_strides_0, {output_strides_0});
14   const ::lightning::Array<float, 1> input(input_ptr - input_offset_0 * input_strides_0, {input_strides_0});
15
16   // Call user kernel
17   stencil(virtual_block_index, n, max_diff, output, input);
18 }
```

Fig. 8: Example of the generated wrapper kernel used internally by Lightning at runtime for Fig. 7.

```
1  let stencil = CudaKernelDef::from_file("stencil.cu")
2    .param_value("n", DTYPE_INT)
3    .param_array("output", DTYPE_FLOAT)
4    .param_array("input", DTYPE_FLOAT)
5    .annotate("global i => read input[i-1:i+1],
6                           write output[i]")
7    .compile(context)?;
8
9  let devices = context.system().devices();
10 let n = 1_000_000;
11 let data_dist = StencilDist::new(64_000, 1, devices);
12 let input = context.ones(n, data_dist)?;
13 let output = context.zeros(n, data_dist)?;
14
15 let work_dist = BlockDist::new(64_000, devices);
16 for _ in 0..10 {
17  stencil.launch(n, 16, work_dist, (n, output, input))?;
18  swap(input, output);
19 }
20
21 context.synchronize()?;
```

Fig. 9: Host code sample for the stencil kernel (Fig. 7).

the full array can be indexed, offsets must be subtracted from the *global* array indices to obtain the *local* chunk indices. To solve this, Lightning uses special data types that subtract an offset from the chunk's memory address. These data types subtract this offset once on construction, meaning that there is no performance cost on element access.

Fig. 8 shows the wrapper kernel generated by Lightning internally at runtime for Fig. 7. This code is shown here for academic purposes, it is not intended to be seen by the end-user. Lines 7-9 show generated constants that are specific for one worker. Lines 12-14 show how the virtual block indices (add offsets) and data types (subtracts offsets) are constructed. Line 17 calls the user's kernel with the correct arguments.

F. Host Code Sample

Fig. 9 shows an example of the host application for the stencil kernel from Fig. 7. Lightning's runtime system is implemented in the Rust programming language. For now, host code also needs to be Rust, but library bindings for other programming languages are part of future work.

First, the kernel source code must be loaded for runtime compilation. Line 1 loads the CUDA kernel code from a separate file `stencil.cu` (shown in Fig. 7), lines 2-6 provide the definition of the kernel's signature (i.e., parameters and data annotations), and line 7 submits the kernel code to the workers for compilation.

Next, the data distributions and arrays must be defined. Line 11 defines the data distribution to be used: a stencil distribution with a chunk size of $64\,000$ (256kB) distributed round-robin across all GPUs. Lines 12-13 define two vectors of size `n` having the above data distribution.

Finally, distributed kernels launches can be submitted. Line 16 defines the superblock distribution to be used: a block distribution having $64\,000$ threads per superblock. Line 17 launches the stencil kernel 10 times with the provided superblock distribution. Kernel launches are asynchronous to the driver, so line 21 blocks the driver until work completes.

IV. Experimental Evaluation

In this section, we present performance results for Lightning. Sec. IV-A describes the experimental setup. Secs. IV-B to IV-E present eight benchmarks on three platforms: one node with one GPU, one node with 4 GPUs, and a cluster with 32 GPUs. Sec. IV-F presents a full application for geospatial cluster analysis that was ported to Lightning

A. Experimental Setup

We performed experiments at Microsoft Azure US East on nodes of type `NC24rsV2`. Each node contains an Intel E5-2690 CPU with 24 cores, 448 GB of memory, 3TB of temporary SSD storage, and 4 NVIDIA Tesla P100 GPUs with 16 GB memory each. The GPUs likely utilize PCIe 3.0 x16 (indicated by bandwidth benchmarks [6]) and nodes are connected to each other by InfiniBand FDR, providing high bandwidth. The software used was Ubuntu 20.04, Rust 1.56, CUDA 11.4, and OpenMPI 4.0.3.

Presented execution times are the average over 5 runs. One initial untimed run is always performed to warm up the system. Each run is measured from the moment that the first distributed kernel launch is submitted until the moment that the driver

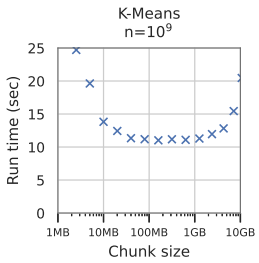


Fig. 10: Throughput versus chunk size for one GPU. Note the logarithmic x-axis.

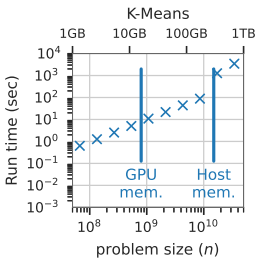


Fig. 11: Run time versus problem size for one GPU. Note the logarithmic axes.

signals the application that all workers finished. This timing thus includes the overhead for execution plan construction. We emphasize that the code is not changed when moving between different platforms.

B. Benchmarks

To evaluate the performance of Lightning in different scenarios, we selected eight CUDA kernels representing different workloads. The kernels were taken from various sources and adapted to make them suitable for Lightning (similar to the example in Fig. 7). The first four benchmarks are compute-intensive (i.e., high arithmetic intensity), while the latter four are data-intensive. For each benchmark, we define a parameter n (the *problem size*) such that the amount of *work* scales linearly with n. However, it is important to note that the amount of *data* need not necessarily scale linearly with n.

- **MD5** (from SHOC [2]) calculates n MD5 hashes in parallel. Work is divided into superblocks of 5B threads each. No data is involved (except one search hash), thus this is a purely compute-oriented benchmark.
- **N-Body** (from CUDA samples [26]) performs 10 iterations of an all-pair gravitational simulation. The benchmark generates $\sqrt{n}$ bodies, so the number of pair-wise interactions (i.e., workload) equals n. The data is replicated (data size is small) and the work is divided equally.
- **Correlator** (from van Nieuwpoort et al. [3]) calculates the correlation between each pair of 256 radio antennas for n frequency channels. The data/work is partitioned with 64 frequency channels per chunk. Note that the original code used a 2D grid of threads and mapped each 2D thread index to a 3D index. This access pattern could not be expressed using Lightning's annotations, thus the code was simplified to use a 3D thread grid instead.
- **K-Means** (from Rodinia [1]) is an iterative clustering algorithm commonly used in data mining. The benchmark uses n records (each having 4 features), finds k=40 clusters, and performs 5 iterations. The distribution uses 25M records per chunk. The original code performed the center calculation on the CPU, but our code utilizes the GPU thanks to Lightning's support for reductions.
- **HotSpot** (from Rodinia [1]) models thermal simulation of an integrated circuit by performing 10 iterations of a 3×3 stencil. The benchmark uses a $\sqrt{n} \times \sqrt{n}$ grid (total of n grid points) with a column-wise distribution such that each chunk contains 50M points. Halo elements are exchanged in each iteration.
- **GEMM** (handwritten, based on Volkov et al. [27]) performs a dense matrix-matrix multiplication $C = AB$. Matrices A, B, and C have size $\sqrt[3]{n} \times \sqrt[3]{n}$ to ensure the total workload is n (cubic time complexity). The matrices are partitioned row-wise with 250M elements per chunk. The work partitioned in the same way, meaning that the data for A and C is available locally, but the entire matrix B must be exchanged between GPUs, making this a very communication-intensive benchmark.
- **SpMV** (from SHOC [2]) performs repeated multiplication of a sparse $\sqrt{n} \times \sqrt{n}$ matrix with a dense vector of size $\sqrt{n}$. Ten iterations are performed, where the output of each operation is used as the input for the next iteration. The vector is broadcast after each iteration. The matrix is stored in ELL format and its density is 0.1% (i.e., the fraction of non-zeros). The vectors are replicated while the matrix is row-wise distributed with 100M elements per chunk.
- **Black-Scholes** (from CUDA samples [26]) computes call-put prices of n financial options using the Black-Scholes model. This problem is embarrassingly parallel since n models can be calculated in parallel. Each chunk contains 100M options.

C. Single GPU

In this section, we present results when using a single GPU. To understand the sensitivity of performance to the chunk size, we evaluated the `K-means` application for different chunk sizes for a problem size that just exceeds GPU memory (n=10^9). The results in Fig. 10 show that the chunk size should not be too small (i.e., $<50MB$, leads to scheduling overhead) or too big (i.e., $>5GB$, prohibits overlapping data transfers and kernel execution). However, a wide range of chunk sizes gives similar performance indicating that performance is not sensitive to the chunk size.

Next, we consider different problem sizes. For example, Fig. 11 shows the execution time versus the problem size for `K-Means`. As anticipated, the run time scales linearly with the problem size n. To ease further analysis, we define *throughput* as the problem size divided by the execution time (i.e., number of items processed per second). Note that the definition of the *problem size* differs per benchmark, thus throughputs are not comparable across benchmarks.

Fig. 12 shows this throughput metric for different problem sizes for each of the eight benchmarks. Nearly all benchmarks show that the throughput is roughly consistent across different problem sizes as long as data fits into GPU memory. This is expected since the workload of each benchmark scales linearly with n. One exception is `SpMV` which performs better for smaller problem sizes. Further examination revealed that this is due to cache behavior since this benchmark involves random accesses and data fits better into caches for smaller n.

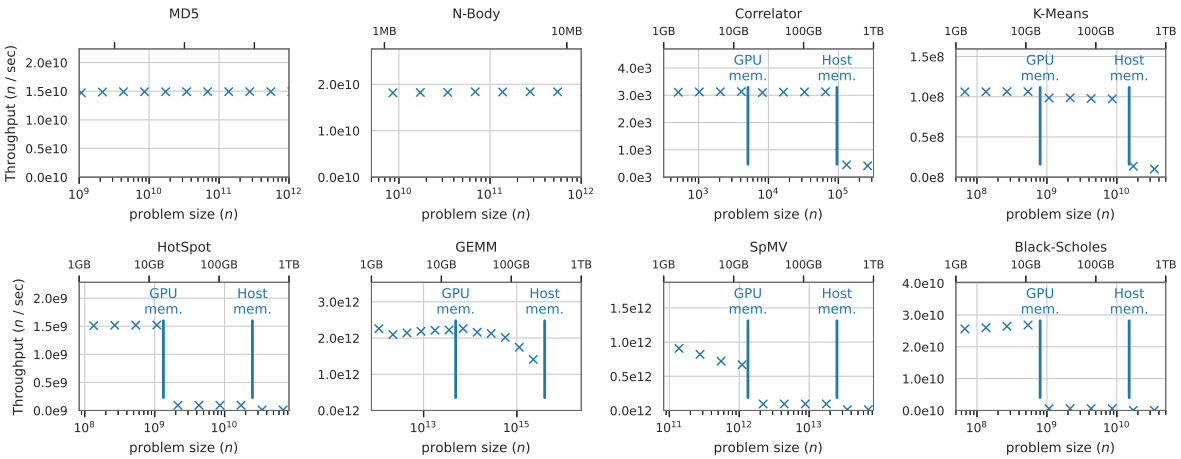


Fig. 12: Throughput versus problem size when using a single GPU. Two vertical lines indicate the largest problem that fits into GPU memory and host memory (`N-Body` and `MD5` always fit). The bottom axis shows problem size (n) while the top axis shows the corresponding memory footprint. Note the logarithmic scale on the x-axis.

Each plot also shows vertical bars indicating the largest problem that fits into GPU memory (first bar) and host memory (second bar). `MD5` and `N-Body` always fit into GPU memory. For large problems, Lightning must spill chunks to host memory (or even disk) and transfer them back to GPU memory as needed, which incurs a performance hit. We see that spilling to disk is never worthwhile due to limited disk bandwidth. It is possible that faster non-volatile storage could make this useful over the regular SSDs in this node.

Spilling to host memory, on the other hand, is beneficial for three benchmarks: `Correlator`, `K-means`, `GEMM`. For these benchmarks, Lightning can overlap kernel execution with the data transfers between GPU and host memory. For example, for `Correlator`, throughput drops by just 8.8% from $n = 16384$ (8.6 GB) to $n = 32768$ (17.2 GB). However, for the three data-intensive benchmarks (`HotSpot`, `SpMV`, `BlackScholes`), overlapping kernel execution with data transfers is not possible since these applications do not perform sufficient work per byte transferred of the PCIe bus. For example, for `BlackScholes` with $n = 0.5 \times 10^9$, the dataset of 10.7 GB is processed in 20.2 ms, meaning that PCIe should provide a bandwidth of 530 GB/s to keep up, over an order of magnitude more than what PCIe 3.0 x16 is capable of.

We conclude spilling to host memory is beneficial for compute-intensive applications. For data-intensive applications, the PCIe bus provides insufficient bandwidth to overlap data transfers. We can avoid spilling by using multiple GPUs since this provides more (combined) GPU memory.

D. Multiple GPUs

Next, we present results when using multiple GPUs on a single node. Fig. 13 shows the throughput for up to 4 GPUs for different problem sizes. Ideally, the throughput should p times higher for p GPUs (i.e., speedup of p). To give an indication of speedup, the labels on the right indicate multiples of the *baseline throughput* (i.e., throughput obtained using one GPU for the largest problem size that still fits into GPU memory).

The plots show that Lightning obtains excellent speedups for all benchmarks. For example, for `Correlator`, `K-means` and `MD5`, speedups are nearly perfect: these benchmarks are compute-intensive and thus scale well. For other benchmarks, such as `GEMM` and `N-Body`, speedups are good except for smaller problem sizes. These benchmarks involve communication, leading to synchronization overhead for small inputs.

Multiple GPUs mean more (combined) GPU memory, indicate in Fig. 13 by the vertical bars that move further to the right as more GPUs are utilized. Larger problems can be processed before data is spilled to host memory. The benchmarks for which spilling was not beneficial in the previous section (`HotSpot`, `SpMV`, and `BlackScholes`) can now scale to larger problems sizes.

However, we also observe that for `Correlator` and `K-means`, for which spilling was beneficial on one GPU in the previous section, spilling is no longer beneficial when using multiple GPUs. For example, for `K-Means`, the throughput on 1 GPU and 2 GPUs is identical for large problems. This happens because GPUs share the PCIe bus, thus using multiple GPUs reduces the effective PCIe bandwidth per GPU. Using multiple nodes circumvents this issue, allowing benchmarks to scale even further.

E. Multiple Nodes

Now, we present the results when using multiple nodes. Fig. 14 shows the throughput for up to 4 nodes with one GPU per node. This figure looks similar to Fig. 13 since both use up to 4 GPUs, except here the GPUs are located on different nodes instead of one node. The most notable difference is that `Correlator` and `K-means` can now scale to larger problem sizes that no longer fit into GPU memory since using multiple nodes means that GPUs no longer share the PCIe bus. These benchmarks are not affected by the network overhead since

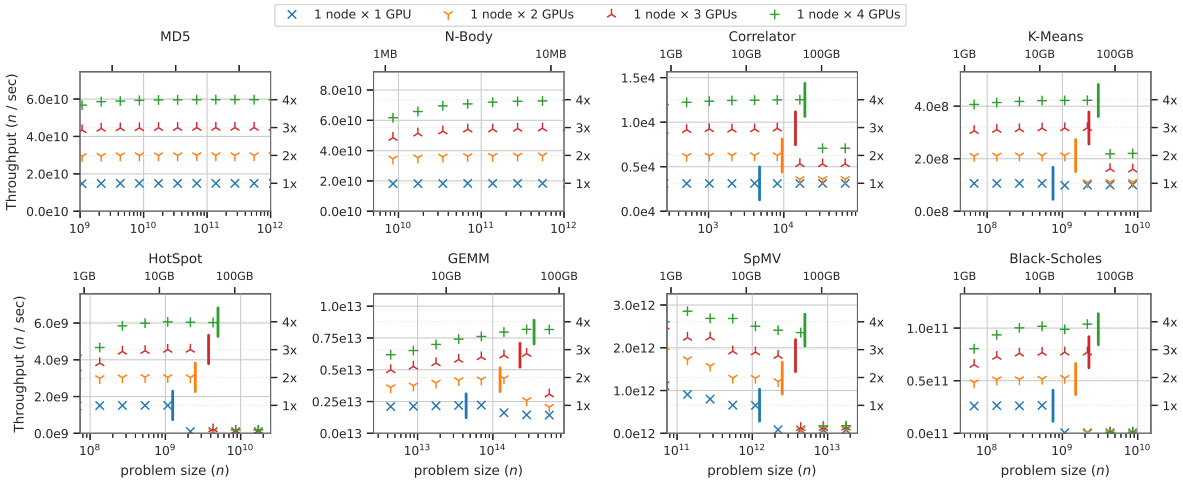


Fig. 13: Throughput versus problem size when using a multi-GPU node. The left labels indicate throughput, bottom labels indicate problem size (n), top labels indicate memory footprint, right labels indicate multiples of the baseline throughput.

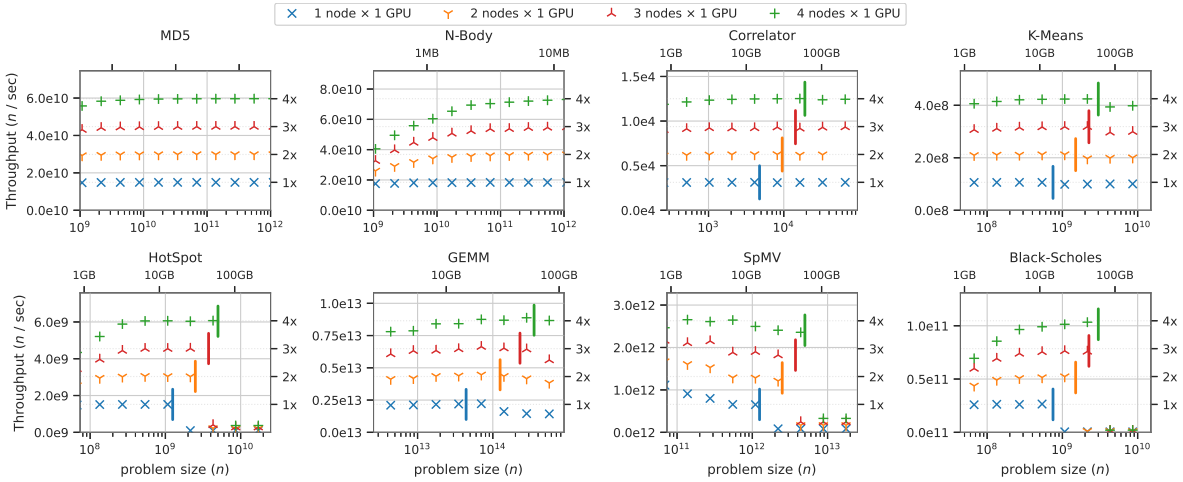


Fig. 14: Throughput versus problem size when using multiple nodes (one GPU per node).

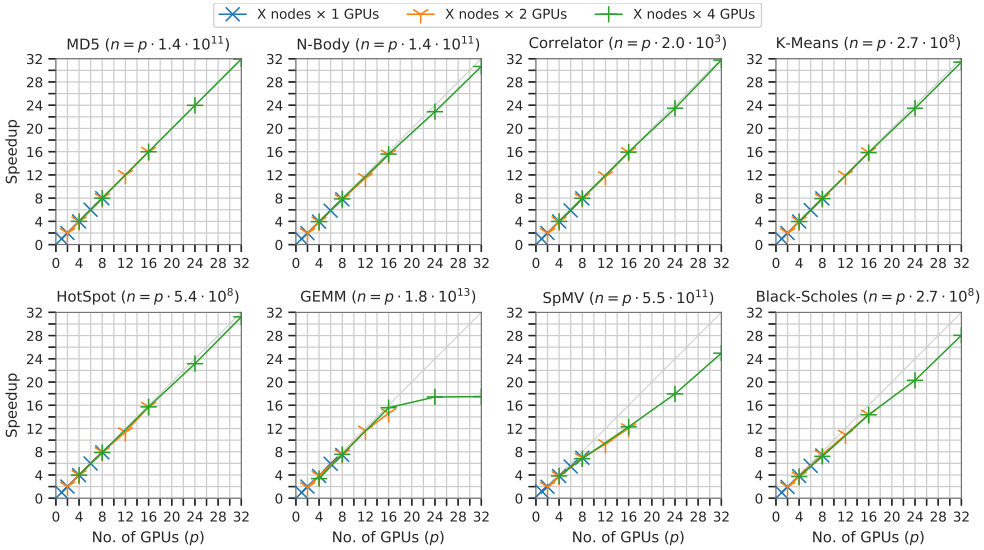


Fig. 15: Weak scaling experiment. Speedup versus number of GPUs (p) for 1, 2, or 4 GPUs per node.

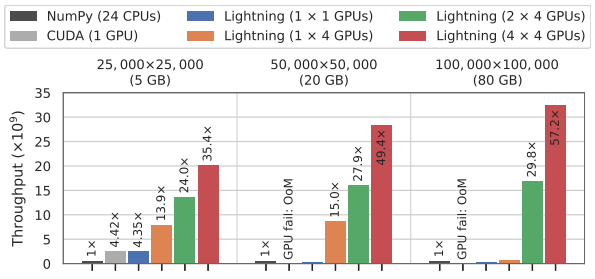


Fig. 16: Performance of application for NumPy, CUDA, and Lightning on three datasets. Throughput is measured as time per iteration divided by matrix size. Notation "$n \times m$ *GPUs*" means n nodes with m GPUs each. OoM is "out of memory".

InfiniBand FDR provides high bandwidth (∼7 GB/s) in the same order as PCIe 3.0 x16 (∼16 GB/s) and Lightning is able to overlap network communication with kernel execution.

Next, we scale to more than 4 GPUs. Fig. 15 shows the speedups up to 32 GPUs using 1, 2, or 4 GPUs per node. For these experiments, we focus on weak scaling, where the problem size n scales according to the number of GPUs p, to emphasize that our framework handles large problems far beyond the capabilities of a single GPU. The results show that `MD5` and `N-Body` scale excellently, which is expected since these benchmarks are compute-intensive and involve little data and communication. `Correlator`, `K-Means`, and `HotSpot` also scale near perfectly, these benchmarks do involve data but there is little communication since GPUs work on their local data. `GEMM` and `SpMV` involve much communication and are more difficult to scale to more nodes. `GEMM` appears to hit the network bandwidth limit at around 16 GPUs. `Black-Scholes`' short run times make scaling difficult. For example, the run time on one GPU is 10.2 ms, while for a 32× larger problem with 32 GPUs the runtime is just 10.8 ms.

F. Full Application

In the previous sections, we considered benchmarks that are simple pipelines of one or two kernels. To evaluate the performance of Lighting for a more complex workflow, we consider the co-clustering algorithm from CGC [28]: a library for geospatial cluster analysis. Co-clustering is an iterative algorithm that clusters the rows and columns of a matrix where these dimensions correspond to space and time. This algorithm can be used, for example, to study the impact of climate change based on the onset of spring in Europe [29]. Each iteration involves three reductions (reduction along the rows, along the columns, and along all entries), leading to a communication-intensive workload on multiple GPUs.

The original code was implemented in Python and accelerated by NumPy. We manually reimplemented this algorithm in CUDA and tuned the resulting 10 CUDA kernels using Kernel Tuner [30], resulting in 635 lines of CUDA code. Next, these kernels were adapted for use in Lightning by modifying 44 lines of code. Fig. 16 shows the performance for NumPy, CUDA, and Lightning for three input matrices: 5 GB (fits into memory of 1 GPU), 20 GB (fits into 4 GPUs), and 80 GB (fits into 16 GPUs). Performance is measured as throughput, i.e., matrix size divided by iteration time.

The results show that for the smallest matrix, the CUDA version is $4.42\times$ faster than the CPU version. Lightning is $4.35\times$ faster, meaning an overhead of just 1.6% over using CUDA directly. This is anticipated since both use the same kernel code, on the same device, for the same dataset. The plots also show that the CUDA version cannot scale to handle the larger datasets that exceed GPU memory. For the largest matrix (80 GB), the CUDA version on one GPU fails while Lightning on 16 GPUs still works and is $57.2\times$ faster than NumPy on the CPU (0.31 sec versus 17.1 sec per iteration).

V. Related Work

GPU programmers have a wide range of options available for creating distributed multi-GPU applications. In general, creating these applications can be achieved by 1) switching to a different programming paradigm or a combination thereof, or 2) using a system that extends the capabilities of the existing GPU programming paradigm.

In the first category, we consider the combination of CUDA/OpenCL with, for example, MPI and OpenMP. We also consider extensions that have been proposed to support GPUs within Big Data frameworks (e.g., Hadoop [23], Spark [24], Dask [22]) or GPU support in common HPC frameworks (e.g., Chapel [31], Charm++ [32], Legion [33], OmpSs [34], PARSeC [35], Global Arrays [36]). The downside of these frameworks is that GPU developers have to learn a new programming paradigm that is different from what they are used to. In addition, while these frameworks give the programmer more control, they also make the programmer responsible for writing complex code to, for example, manage GPU memory, move data, split work into smaller jobs, and overlap computation and communication. Instead, Lightning allows GPU programmers to interact with a multi-GPU cluster as if there existed a single large virtual GPU. In Lightning, programmers can create arrays and launch kernels as they are used to, while the work/data is automatically distributed.

There have been previous studies that also propose extensions to existing GPU programming models (CUDA and OpenCL) to facilitate the creation of distributed or multi-GPU applications. Here, we distinguish two different approaches in the literature: A) Frameworks give explicit control over remote GPUs, and B) frameworks that implicitly distribute work across multiple GPUs.

A. Explicit control over multiple/remote GPUs

There have been several projects that allow remote GPUs to be used as though there were local. Some projects aim at the virtualization of remote GPUs in the context of cloud computing, examples are GridCUDA [9], rCUDA [8], gVirtuS [10], and DS-CUDA [11]. Strengert et al. [37] propose an interesting extension to the CUDA model that extends CUDA's three-level parallelism hierarchy (thread, block, grid) with additional

levels (*bus*, *network*, *application* levels). For OpenCL, there have also been several attempts to provide access to remote devices, for example, SnuCL/SnuCL-D [12], [14], Distributed OpenCL (dOpenCL) [13], clOpenCL [15], LibWater [16], HybridOpenCL [17], EngineCL [18], and dOCAL [38].

However, all the above solutions purposely do not offer any abstraction over the direct CUDA/OpenCL API, meaning programmers must manually divide the work, partition the data, and perform data transfers between GPUs. Lightning, on the other hand, allows programmers to use a cluster of GPUs in a way that resembles single GPU programming.

B. Implicitly scaling to multiple/remote GPUs

There have been a few previous works that attempt to abstract multiple physical GPUs into a single virtual GPU. Kim et al. [19] present a framework that offers multiple GPUs in one node as a single virtual OpenCL device. Launching a kernel onto this virtual device will automatically distribute the workload and transfer the data between host and GPU memory. There are four key aspects in which this work differs from Lighting: 1) only a single node is supported; 2) each array is entirely allocated in host memory which limits scalability; 3) workload is automatically partitioned using heuristics which forbids performance tuning and takes away control from the programmer; 4) access patterns are determined by using runtime sampling which has a runtime overhead, and can lead to misclassification, whereas Lightning's data annotations have no runtime overhead and ask programmers to consider the access pattern of their kernels.

DistCL [20] is another framework that offers multiple GPUs as a single virtual OpenCL device, while also supporting clusters of GPUs. There are three key differences between DistCL and Lightning: 1) each array is entirely allocated in the GPU memory of each device which limits scalability; 2) workloads are always partitioned along the most significant dimension, whereas Lightning allows custom workload distribution policies; 3) DistCL requires the programmer to write special *meta-functions* that indicate intervals accessed by each kernel, whereas Lightning's data annotations present a more intuitive declarative approach.

MAPS-Multi [21] is most closely related to Lightning. MAPS-Multi is a multi-GPU programming system that facilitates workload distribution across multiple GPUs in a single node using a set of predefined data access patterns. Lightning is more flexible, allowing any linear data access pattern. MAPS-Multi requires substantial modifications to CUDA kernel code, for example, for-loops are replaced with custom macros and data needs to be explicitly committed to memory. Lightning, on the other hand, allows for existing CUDA kernels to be reused. MAPS-Multi makes programmers responsible for data synchronization, whereas Lightning automatically takes care of this and overlaps inter-/intra-node communication and GPU computations. Lightning also supports distributed computing over GPUs in multiple nodes, while MAPS-Multi does not.

VI. Conclusions & Future Work

In this work, we presented Lighting: a framework that enables GPU kernels to run on any amount of data and run on any number of GPUs, even across different nodes. Our solution offers abstractions for *distributed kernel launches* and *distributed arrays* that enable transparent distribution of work and data across multiple GPUs. Data annotations allow the framework to infer data requirements and data dependencies. Lightning obtains excellent performance through asynchronous processing by overlapping plan construction, scheduling, data movement, and kernel execution. Lightning is available online as open source software [39][2].

Evaluation shows great results. We observe that spilling to host memory allows data-intensive applications to work on massive data sets. Experiments on four GPUs on a single node show excellent speedups, except spilling becomes less beneficial since GPUs on one node share PCIe bandwidth, which can be overcome by using multiple nodes. Spilling to disk appears to be not beneficial due to limited disk bandwidth, it is possible that faster non-volatile memory (NVM) could provide a solution here. The geospatial clustering application shows that our framework can handle large datasets, for example, processing 80 GB with 16 GPUs is $57.2\times$ faster than the CPU-version. Processing this dataset using one GPU would be impractical in terms of memory and processing power.

There are several avenues for future work. Lightning's model is language-agnostic and support for other languages besides CUDA is in progress (e.g., OpenCL). Additionally, Lightning currently requires manual selection of work/data distributions. We are working on assistance in this selection (e.g., via profiling) or even automatic selection (i.e., more intelligent planner). There are also various interesting future topics that we did not touch upon, such as load-balancing, heterogeneous platforms, and fault-tolerance.

References

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2009. https://doi.org/10.1109/IISWC.2009.5306797 pp. 44–54.

[2] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: Association for Computing Machinery, 2010. https://doi.org/10.1145/1735688.1735702 p. 63–74.

[3] R. V. van Nieuwpoort and J. W. Romein, "Correlating radio astronomy signals with many-core hardware," *Int J Parallel Prog*, vol. 39, 2011. https://doi.org/10.1007/s10766-010-0144-3

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for Large-Scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283.

[5] S. Heldens, P. Hijma, B. V. Werkhoven, J. Maassen, A. S. Z. Belloum, and R. V. Van Nieuwpoort, "The landscape of exascale research: A data-driven literature analysis," vol. 53, no. 2, mar 2020. https://doi.org/10.1145/3372390

[2]https://github.com/lightning-project

[6] B. v. Werkhoven, J. Maassen, F. Seinstra, and H. Bal, "Performance models for cpu-gpu data transfers," in *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014. https://doi.org/10.1109/CCGrid.2014.16 pp. 11–20.

[7] B. van Werkhoven, W. J. Palenstijn, and A. Sclocco, "Lessons learned in a decade of research software engineering gpu applications," in *Computational Science – ICCS 2020*, V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, Eds. Cham: Springer International Publishing, 2020. ISBN 978-3-030-50436-6 pp. 399–412.

[8] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *International Conference on High Performance Computing Simulation*, 2010. https://doi.org/10.1109/HPCS.2010.5547126 pp. 224–231.

[9] T.-Y. Liang and Y.-W. Chang, "GridCuda: A Grid-Enabled CUDA Programming Toolkit," in *IEEE Workshops of International Conference on Advanced Information Networking and Applications*, 2011. https://doi.org/10.1109/WAINA.2011.82 pp. 141–146.

[10] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU Transparent Virtualization Component for High Performance Computing Clouds," in *Proceedings of the 16th International Euro-Par Conference on Parallel Processing*, ser. EuroPar'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 379–391.

[11] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi, "DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment," in *SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012. https://doi.org/10.1109/SC.Companion.2012.146 pp. 1207–1214.

[12] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: Association for Computing Machinery, 2012. https://doi.org/10.1145/2304576.2304623 p. 341–352.

[13] P. Kegel, M. Steuwer, and S. Gorlatch, "dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems," in *IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, 2012. https://doi.org/10.1109/IPDPSW.2012.16 pp. 174–186.

[14] J. Kim, G. Jo, J. Jung, J. Kim, and J. Lee, "A Distributed OpenCL Framework Using Redundant Computation and Data Replication," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: Association for Computing Machinery, 2016. https://doi.org/10.1145/2908080.2908094. ISBN 9781450342612 p. 553–569.

[15] A. Alves, J. Rufino, A. Pina, and L. P. Santos, "clOpenCL - Supporting Distributed Heterogeneous Computing in HPC Clusters," in *Euro-Par: Parallel Processing Workshops*, I. Caragiannis, M. Alexander, R. M. Badia, M. Cannataro, A. Costan, M. Danelutto, F. Desprez, B. Krammer, J. Sahuquillo, S. L. Scott, and J. Weidendorfer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 112–122.

[16] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer, "LibWater: Heterogeneous Distributed Computing Made Easy," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: Association for Computing Machinery, 2013. https://doi.org/10.1145/2464996.2465008. ISBN 9781450321303 p. 161–172.

[17] R. Aoki, S. Oikawa, R. Tsuchiyama, and T. Nakamura, "Hybrid OpenCL: Connecting Different OpenCL Implementations over Network," in *10th IEEE International Conference on Computer and Information Technology*, 2010. https://doi.org/10.1109/CIT.2010.457 pp. 2729–2735.

[18] R. Nozal, J. L. Bosque, and R. Beivide, "EngineCL: Usability and Performance in Heterogeneous Computing," *Future Generation Computer Systems*, vol. 107, pp. 522–537, 2020. https://doi.org/10.1016/j.future.2020.02.016

[19] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a Single Compute Device Image in OpenCL for Multiple GPUs," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: Association for Computing Machinery, 2011. https://doi.org/10.1145/1941553.1941591 p. 277–288.

[20] T. Diop, S. Gurfinkel, J. Anderson, and N. E. Jerger, "DistCL: A Framework for the Distributed Execution of OpenCL Kernels," in *IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, 2013. https://doi.org/10.1109/MASCOTS.2013.77 pp. 556–566.

[21] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin, "Memory access patterns: the missing piece of the multi-GPU puzzle," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015. https://doi.org/10.1145/2807591.2807611

[22] RAPIDS - Open GPU Data Science, "Dask-CUDA." [Online]. Available: https://docs.rapids.ai/api/dask-cuda/nightly/index.html

[23] J. Zhu, J. Li, E. Hardesty, H. Jiang, and K.-C. Li, "GPU-in-Hadoop: Enabling MapReduce across distributed heterogeneous platforms," in *IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)*, 2014. https://doi.org/10.1109/ICIS.2014.6912154 pp. 321–326.

[24] Y. Yuan, M. F. Salmi, Y. Huai, K. Wang, R. Lee, and X. Zhang, "Spark-GPU: An accelerated in-memory data processing engine on clusters," in *IEEE International Conference on Big Data (Big Data)*, 2016. https://doi.org/10.1109/BigData.2016.7840613 pp. 273–283.

[25] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. C-28, no. 9, 1979.

[26] "CUDA C++ Programming Guide," http://docs.nvidia.com/cuda/.

[27] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008. https://doi.org/10.1109/SC.2008.5214359

[28] F. Nattino, O. Ku, M. W. Grootes *et al.*, "Clustering Geo-Data Cubes (CGC): A Clustering Tool for Geospatial Applications," Sep. 2021. https://doi.org/10.5281/zenodo.5524610

[29] X. Wu, R. Zurita-Milla, and M. Kraak, "A novel analysis of spring phenological patterns over Europe based on co-clustering," *Journal of geophysical research: Biogeosciences*, vol. 121, no. 6, pp. 1434–1448, 2016. https://doi.org/10.1002/2015JG003308

[30] B. van Werkhoven, "Kernel tuner: A search-optimizing gpu code auto-tuner," *Future Generation Computer Systems*, vol. 90, pp. 347–358, 2019. https://doi.org/10.1016/j.future.2018.08.004

[31] A. Hayashi, S. R. Paul, and V. Sarkar, "GPUIterator: Bridging the Gap between Chapel and GPU Platforms," in *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*, ser. CHIUW 2019. New York, NY, USA: Association for Computing Machinery, 2019. https://doi.org/10.1145/3329722.3330142. ISBN 9781450368001 p. 2–11.

[32] R. Vasudevan, S. S. Vadhiyar, and L. V. Kalé, "G-Charm: An Adaptive Runtime System for Message-Driven Parallel Applications on Hybrid Systems," in *27th International ACM Conference on International Conference on Supercomputing - ICS '13*, 2013. ISBN 978-1-4503-2130-3

[33] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012. https://doi.org/10.1109/SC.2012.71

[34] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, "Productive Programming of GPU Clusters with OmpSs," in *IEEE 26th International Parallel and Distributed Processing Symposium*, 2012. https://doi.org/10.1109/IPDPS.2012.58 pp. 557–568.

[35] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra, "Hierarchical DAG Scheduling for Hybrid Distributed Systems," in *IEEE International Parallel and Distributed Processing Symposium*, 2015. https://doi.org/10.1109/IPDPS.2015.56 pp. 156–165.

[36] V. Tipparaju and J. S. Vetter, "Ga-gpu: Extending a library-based global address spaceprogramming model for scalable heterogeneouscomputing systems," in *CCF*. ACM, 2012.

[37] M. Strengert, C. Müller, C. Dachsbacher *et al.*, "CUDASA: Compute unified device and systems architecture," in *8th Eurographics Conference on Parallel Graphics and Visualization*, 2008. ISBN 978-3-905674-04-0

[38] A. Rasch, J. Bigge, M. Wrodarczyk, R. Schulze, and S. Gorlatch, "DOCAL: High-Level Distributed Programming with OpenCL and CUDA," *J. Supercomput.*, vol. 76, no. 7, p. 5117–5138, jul 2020. https://doi.org/10.1007/s11227-019-02829-2

[39] S. Heldens, "Lightning: Fast data processing using GPUs on distributed platforms," Feb. 2022. https://doi.org/10.5281/zenodo.6281459. [Online]. Available: https://github.com/lightning-project/lightning