Autonomous Execution for Multi-GPU Systems: Compiler Support

Javid Baydamirli
Koç University
Istanbul, Turkey
jbaydamirli21@ku.edu.tr

Tal Ben Nun
Lawrence Livermore National Laboratory
Livermore, California, USA
bennun2@llnl.gov

Didem Unat
Koç University
Istanbul, Turkey
dunat@ku.edu.tr

***Abstract*—Recent trends in HPC systems increasingly emphasize accelerators, particularly GPUs, as autonomous execution units, shifting control of entire program execution to GPUs. Communication libraries enable devices to move data independently among one another, bringing forth latency improvements, and first-party GPU runtimes expose APIs for kernels to organize their execution. Despite the trends and advancements, current high-level frameworks and compilers lack support for constructs enabling this *autonomous execution*. In this work, we aim to bridge this gap with a compiler and provide a productive method for writing efficient GPU-first code. We design and develop a code generator that efficiently fuses and schedules persistent kernels, provides high-level abstractions over device resources, and enables GPU-initiated communication within Python code using NVSHMEM to realize autonomous multi-GPU execution. We compare our implementation to other accelerated Python compilers including CuPy, DaCe, and cuNumeric on 22 NPBench kernels. We additionally perform a scaling study of distributed 2D/3D Jacobi and observe a speedup of $6.1x$ and $30.8x$ over DaCe and cuNumeric, respectively, on 8 GPUs for the 3D case with a scaling efficiency of 98%.**

***Index Terms*—Multi-GPU, GPU-initiated communication, NVSHMEM, Python**

I. Introduction

The proliferation of GPU accelerators in top systems has brought forward changes in multi-GPU programming paradigms as many applications are now scaled up to several nodes and consequently bound by communication and synchronization overheads. One such notable shift has been in the form of new execution models wherein the control path is moved to devices, granting them more autonomy over their computation *and* communication among peer GPUs. Prior work has explored various degrees of such autonomy for several use cases, including irregular computations [1], [2], graphics workloads [3], execution models [4] GPU-initiated communication [5]–[10], and GPU-triggered networking [11], seeing improved overheads compared to traditional execution, as many of the host-induced latencies are eliminated or reduced.

There is now first-party support by the two big vendors, NVIDIA and AMD, for GPU-initiated communication through optimized libraries, NVSHMEM [6] and ROC_SHMEM [12], respectively, that facilitate data transfers in both single and multi-node systems with fine-grained device-side APIs. Both libraries have seen use in earlier research [4]–[9], and been adopted into several parallel frameworks [10], [13]–[15]. Finally, recent work by [4] demonstrates the efficacy of fully autonomous execution in Stencil and Conjugate Gradient mini-applications in NVIDIA GPUs and presents a blueprint for *CPU-free* programs. The work combines several preexisting techniques such as persistent kernels and device-side communication to achieve significant latency improvements on a single node.

However, writing multi-GPU code in an emerging programming model can be a daunting and error-prone process that requires familiarity with low-level constructs and architecture specifics. This lack of productivity is coupled with the vendor-dependence of first-party GPU programming toolkits, CUDA and ROCm, and their respective communication libraries, leading to poor, if any, performance portability. Several works have addressed these issues – frameworks such as Kokkos [16], RAJA [17], and SYCL are lower-level libraries that focus on performance portability, while other bodies of work have put efforts into expressive DSLs and high-level tools. Python in particular is a popular choice of language for such targets thanks to its high-level productive nature and extensive repertoire of fast numerical libraries. Though full-scale distributed applications can be written in Python with the help of the abstractions provided by these frameworks such as DaCe [18] and Dask [19], support for autonomous execution has been absent. This is the case in both the *API* and the *backend*.

Firstly, code written by the users cannot utilize GPU-centric constructs and libraries. While GPU vendors have increased their support for more device autonomy and work has been put into employing devices as independent processors, GPU constructs such as thread blocks are not exposed to the user and they are instead treated as large bulk-computation devices. This causes a disconnect of user code from the actual structure of devices as GPU-centric optimizations cannot easily be expressed. Moreover, the choice for the communication method is limited to the traditional, host-initiated libraries, such as `mpi4py`.

Secondly, in the *backend*, code generated by the frameworks, internally, do not make use of GPU-centric concepts and libraries. Support for device-initiated communication is again mostly absent - among the most similar related work, DaCe relies exclusively on MPI routines for its communication

layer, while Dask utilizes MPI and NCCL, both of which have host-side APIs and require host-device synchronization.

In this paper, we address these gaps and develop a code generator for a CPU-free autonomous execution model. In short, our paper contributes the following:

- We provide a Python-based API to write CPU-free code with GPU-centric constructs and GPU-initiated communication calls.
- We implement a code-generation backend that creates persistent kernels and schedules threads and thread blocks to maximize efficiency.
- We develop an autonomous communication layer using `NVSHMEM` and `ROC_SHMEM` as Pythonic abstractions to be used on the frontend. To our knowledge, this is the first body of work enabling GPU-initiated communication directly in Python code.
- We conduct a study to measure the performance of our implementation on 22 kernels from the NPBench [20] suite on a single GPU and compare the performance against DaCe and CuPy. Additionally, we compare the performance of 2D and 3D multi-GPU stencil computation against multi-GPU DaCe and cuNumeric.

To avoid rebuilding the entire code generation framework, we opt to leverage an existing Python framework, DaCe, and extend it to support the autonomous execution model.

II. Background

In a traditional GPU execution model, the host CPU acts as the orchestrator of the execution, launching discrete kernels to offload computation to the device. Synchronization is host-driven as well with synchronization points inserted in the host code to ensure GPU operations are completed before proceeding. Streams managed on the host side provide the means to overlap computation and communication. Multi-GPU execution follows a similar pattern, with the host controlling data transfers between GPUs and synchronizing their execution.

Autonomous execution, unlike traditional host-driven execution, involves the host CPU only at the initial kernel launch, after which control and data paths are managed on the GPU side. The GPU handles computation and communication overlap, as well as intra-device and inter-device synchronization, independently of the host. This section introduces the necessary concepts that are prerequisites to our implementation.

A. Persistent Kernels

GPU kernels have traditionally been implemented in a bulk-synchronous manner - also referred to as *discrete kernels* [21]. For instance, an iterative solver kernel is scheduled on a per-iteration basis, getting torn down and relaunched for every time step. Each instance of a GPU kernel is only concerned with a specific portion of the computation, and unaware of the underlying iterative structure of the application, as well as possible communication routines enqueued in concurrent streams.

Ismayilov et al. [4] explore such iterative workloads and provide GPUs in a single node more autonomy by making use of persistent kernels [22] where the time loop is moved inside the kernel, resulting in a single kernel launch for the entirety of the application. Though not inherently more performant in all cases [22]–[24], there are wider implications of persistent execution, especially in multi-GPU scenarios where communication has traditionally been initiated outside of devices, as discussed below.

B. Device-Side Synchronization

In-kernel synchronization had been limited to a single thread block at most prior to CUDA 9.0, which introduced the Cooperative Groups API, allowing more granular synchronization of threads as well as introducing a global barrier. By itself, the latency difference between implicit synchronization using repeated kernel launches and explicit synchronization is negligible [24], however, maintaining a single kernel throughout the computation is desirable, as more caching optimizations and better shared memory utilization [23], whose lifetime ends with the kernel, is possible thanks to the kernel not being destroyed after each time step.

Moreover, similar to implicit kernel synchronization within a single GPU, barriers among peer devices in discrete multi-GPU kernels are also managed by the CPU, through host-side barriers provided by interfaces such as OpenMP and MPI.

C. Thread Block (TB) Specialization

It is often necessary for multi-GPU applications to overlap their computation with communication to hide the latency of the latter to achieve optimal performance, as data transfers have a high cost. In discrete kernels, this is commonly achieved through multiple kernel launches across asynchronous GPU streams. Though neither CUDA nor ROCm guarantee *concurrent* scheduling of streams, they are a convenient method of hiding latency while simultaneously allowing individual kernels to be oversubscribed. Moving to the persistent execution model, however, reduces opportunities for utilizing established stream facilities. As discussed previously, the model necessitates launching a single large kernel encapsulating all concurrent steps, meaning there is no longer any inter-kernel concurrency that can be delegated to the GPU runtime.

Instead, thread blocks in a persistent kernel can be used to establish concurrency as standalone execution units. Prior works [2], [25] explore the concept at warp and block levels and assign concurrent sub-tasks to them to avoid global barriers and address irregular computations. We adapt this concept in our design using the aforementioned Cooperative Groups facilities that allow more control over the scheduling of thread blocks.

D. GPU-Initiated Communication

Current methods of communication across GPUs make use of peer-to-peer data paths to move data [26], [27], utilizing the high bandwidth of GPU interconnects and avoiding extraneous

buffers. MPI is a particularly popular communication standard with support for device pointers for both CUDA [28], [29] and ROCm [30] platforms. Despite its wide applicability, the MPI model is fundamentally dissimilar to GPU programming models, notably lacking any knowledge of GPU streams and requiring explicit host synchronization of streams, which in turn makes pipelining less effective [14], [31]. Alternative communication libraries have been created by GPU vendors - NCCL and RCCL by NVIDIA and AMD, respectively. Though they expose a similar send-receive API, the two libraries, in contrast to MPI, *are* aware of GPU streams and include on-stream versions. Both libraries have seen a great amount of adoption in deep learning frameworks thanks to their fast collective operations.

1) GPU-Shmem: More recently, PGAS-based libraries, NVSHMEM [32] and ROC_SHMEM [12] have taken GPU communication a step further by allowing communication to be issued directly from within kernels. Semantically, this model of communication is the most compatible with the massively parallel, relaxed-memory nature of GPUs, as they move data with one-sided remote read and writes, analogous to the GPU memory model without the complexities of message handling.

The basis of communication in NVSHMEM is the one-sided `put/get` methods, along with collective operations that may include the entirety or a user-defined subset of the processing elements (PEs). The methods are each further split into blocking and non-blocking variants. It should be noted that since the NVSHMEM memory model is highly relaxed, blocking `put` variants do not guarantee the delivery to remote PEs, or the ordering at the local PE; they simply return after data has been copied out of the caller [32]. As such, extra care must be taken when ordering is required using the `fence` and `quiet` operations to enforce ordering.

NVSHMEM additionally provides fine-grained synchronization with signaling operations. For variants that operate on contiguous memory, the library provides *composite* API calls that perform memory operations and set a given signal as an atomic operation. It should be noted that using synchronization is only possible in persistent kernels to avoid deadlocks [32].

In addition, both GPU-Shmem libraries have support for native GPU groups - thread blocks and warps - where remote memory and synchronization calls can be invoked collectively by a block of a warp. We make use of this feature in tandem with thread block specialization.

III. Compiler for Autonomous Execution

We aim to bridge the gap between autonomous GPU execution and high-level Python code in the form of a compiler. We give an overview of our approach in Figure 1 consisting of two parts: the user-facing API, and the implementation at the compiler side.

First, on the Python side, we introduce abstractions over the processing elements of devices, namely thread blocks, allowing the user to optionally manage them. We then enable support for GPU-initiated communication from the Python API with high-level range-based indexing operations and discuss our approach in greater detail in Section III-B.

Secondly, on the compiler side, we introduce storage for communication buffers and present our communication scheduling strategy. For computation, we detail our scheduling strategy for threads and thread blocks, discuss our approach to persistent fusion, and outline the structure of the generated code in Sections III-A and III-C. Our compiler is built on top of the `DaCe` framework with the Stateful DataFlow multi-Graph (SDFG) IR [18], to which we apply transformations and code generation techniques to generate CPU-free code.

A. Thread Scheduling

While traditional discrete kernels provide a natural mapping of computational resources - threads and thread blocks - to data through kernel launch parameters, persistent execution, as mentioned previously, imposes certain limitations that must be considered in the generated code. The benefit and purpose of discrete kernel parameters in this context is twofold: Firstly, kernels of this kind can be launched with a practically infinite number of thread blocks regardless of the real hardware capabilities of the device. A smaller subset of blocks that can run concurrently in the device are then scheduled automatically to the processing units in the GPU by the hardware. This allows the programmer to write efficient kernels that are oblivious to the physical configuration of the GPU using *virtual* blocks. Correspondingly, a code-generating compiler can set the kernel parameters in accordance to the data at hand, simplifying the resulting device code.

Switching to persistent execution, however, puts constraints on kernel launch parameters, limiting them to the maximum occupancy of the device. While this constraint resigns control over the execution elements of the device from the runtime it necessitates extra care by the compiler to efficiently schedule them, which we will discuss next.

To aid our discussion, we will use the Softmax kernel. Softmax is a common function used in many deep learning methods. The numerically stable version of the function introduced in Listing 1 consists of a max reduction of the input along its last axis, which is then subtracted from the input and exponentiated. The result is then normalized by dividing it by its max reduction.

The Softmax kernel illustrates two challenges in CPU-free code generation: (i) High-dimensionality of the data requires upscaling the grid, (ii) persistent fusion of steps with different sizes requires partitioning blocks. How the compiler overcomes these challenges will be discussed in Section III-A1 and III-A2, respectively.

1) Upscaling the grid and multi-dimensional maps: The first challenge we face in persistent scheduling is efficiently mapping threads to large domains and high-dimensional data. In order to have the entire device active, kernels generated by the compiler are launched with the maximum number of blocks in a 1D configuration. Since this kind of kernel configuration creates linear block indices, the compiler needs

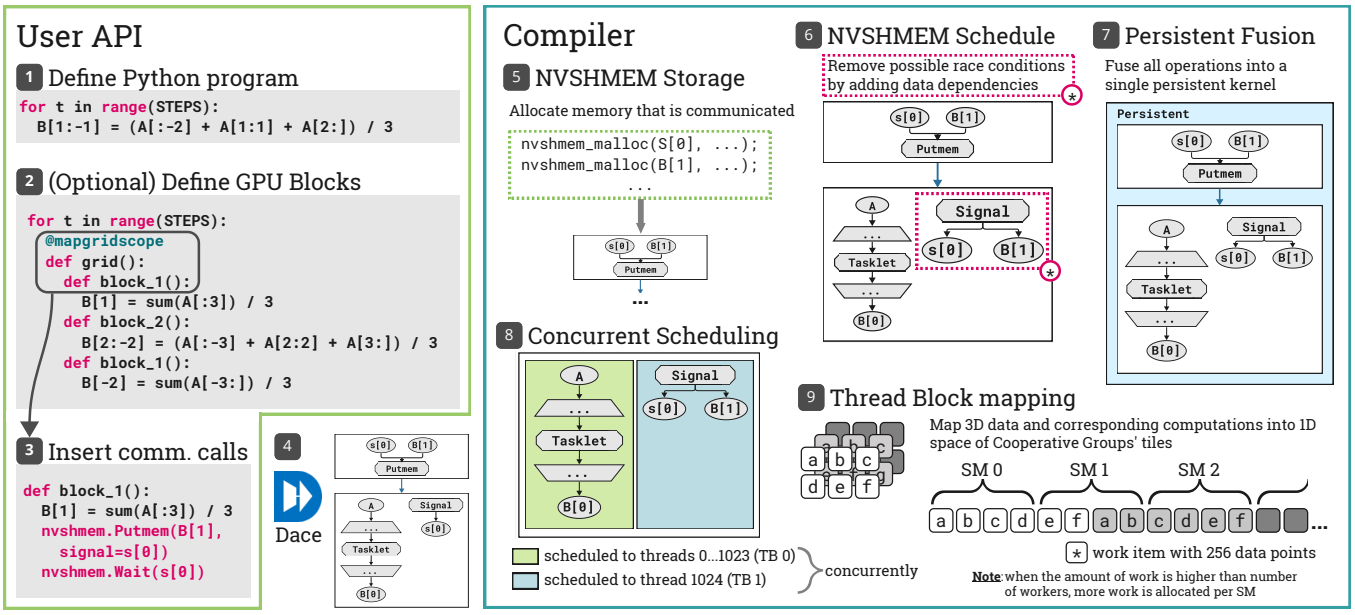

Fig. 1: Overview of compilation for multi-GPU autonomous execution: Python API, and Backend

to reshape them as needed.

```
def softmax(x: np.float64[N, C, H, W]):
    tmp_max = np.maximum.reduce(x, axis=-1, keepdims=True)
    tmp_out = np.exp(x - tmp_max)
    tmp_sum = np.add.reduce(tmp_put, axis=-1,
          keepdims=True)

    return tmp_out / tmp_sum
```

Listing 1: Softmax kernel

For example, consider the Softmax kernel in Listing 1. The kernel accesses the 4-dimensional array, `x[N, C, H, W]` at multiple axes simultaneously. In particular, the `np.exp(x - tmp_max)` step requires indexing into the 4th dimension of the array while using values from the 3-dimensional `tmp_max` array *corresponding to the indices of* x. Since we need indices from several dimensions, it is not possible to linearize all axes to fit into 1D blocks. Moreover, naively generating nested loops for each dimension is infeasible, as it would significantly hurt the performance due to a large number of thread blocks sitting idle in smaller-sized nested loops.

We propose a heuristic solution to linearize the domain more intelligently. The compiler first identifies the largest contiguous dimension accessed in the computation; in the Softmax example, this corresponds to the 4^{th} dimension of the input tensor. This dimension is prioritized as the contiguous dimension, and the block is specialized for its size, discussed in the following subsection. Next, the remaining three dimensions, in reverse order, are fused into a block-loop. For Softmax, this would be dimensions `[N, C, H, W]`, with **H** corresponding to the block index. Finally, if more axes remain, they are put into sequential nested loops.

```
for (auto _i = blockidx.x; _i < ((N * C) * H); _i +=
     gridDim.x) {
    auto _i0 = (_i % N);
    auto _i1 = ((_i / N) % C);
    auto _i2 = ((_i / (N * C)) % H);

    for (auto _i3 = threadIdx.x; _i3 < W; _i3 +=
         blockDim.x) {
        tmp_out[... + _i3] = exp(x[... + _i3] + tmp_max[...
            + _i3];
    }
}
```

Listing 2: Fusing nested dimensions of Softmax

Our objective with this scheme is to have a large, fused loop with enough elements to occupy all streaming multiprocessors (SMs) of the GPU while prioritizing parallel regions of the code, and exploiting the remaining parallelism from thread blocks. We note that we found it ineffective to fuse beyond 3 dimensions, as the overhead of computing indices grows to a noticeable degree. Listing 2 shows a sample generated code with fused dimensions and a parallel thread loop within.

2) Downscaling blocks and block-size specialization: Additionally, grid and block configurations of discrete kernels can be specialized to the dimensions of the computed data. Though thread blocks in both CUDA and ROCm have an upper limit of 1024 threads, they can be scaled down to fit to the data more precisely. This has significant performance implications for high-dimensional kernels where the computation is repeated over a contiguous dimension, for which the launch parameters can be tuned. While persistent kernels can similarly be launched with a smaller number of threads in exchange for more thread blocks, it is infeasible to do so when persistently fusing kernels that have computations in different dimensions, on axes, or through unequally shaped data.

Consider the Softmax kernel again in Listing 1. The kernel contains by two reduction steps over the last axis followed by exponentiation, subtraction, and division operations over the

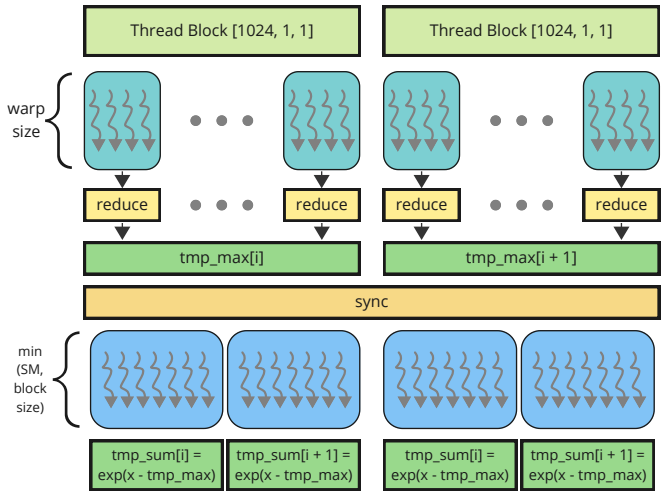

Fig. 2: Partitioning thread blocks in accordance to computation shape of Softmax [N, H, SM, SM]

entire domain. Softmax is characterized by a large contiguous dimension that can be mapped to 1D threads for subtraction and exponentiation steps - performing best with large, one-dimensional blocks, whereas reductions require warp-level primitives for optimal performance [33] - needing a larger number of warp-sized blocks. As we fuse all steps into one persistent kernel, we are unable to specialize the launch parameters for any one step.

To alleviate this, the compiler generates kernels launched with the largest possible block size of `[1024, 1, 1]` and partitions them to smaller chunks as needed, as shown in Figure 2. This can be achieved efficiently without state management thanks to the Cooperative Groups support in CUDA and ROCm that provide facilities to statically subdivide thread blocks. Namely, the `tiled_partition` API allows statically defining equally sized tiles at compile time for each thread block. As described in the previous section, we prioritize the most contiguous dimension to partition the tiles for. Listing 3 shows a sample generated code using this strategy.

```
auto tile = cg::tiled_partition<min(power_2(W),
     gridDim.x)>(block);
for (auto _i = blockidx.x * num_tiles + tile_idx; ...) {
   // Fused dimensions

   for (auto _i3 = tile.thread_rank(); _i3 < W; _i3 +=
        tile.size()) {
      tmp_out[... + _i3] = exp(x[... + _i3] + tmp_max[...
           + _i3];
   }
}
```

Listing 3: Softmax with thread block specialization using partitioned tiles

B. GPU-Initiated Communication Layer

To implement non-trivial multi-GPU programs in a CPU-free manner, we need an autonomous communication layer. While CUDA kernels can issue low-latency communication simply by writing to remote memory pointers, we opt to implement our communication layer exclusively with GPU-shmem libraries. [12], [32]. As discussed previously, they provide a relaxed, one-sided memory model that lends itself to autonomous execution naturally with a convenient device-side API. This section discusses the specifics of how we bridge GPU-side communication calls to Python.

1) Python API and the Communication Model: As per the OpenSHMEM API, GPU-shmem libraries provide two methods for one-sided remote data access: `put`, `get`, and `signaling` operations for synchronization. For simplicity and easier memory management, we focus only on the `put` methods that write to remote devices' memories. As such, the interface we expose for one-sided communication consists of two abstracted calls:

- `Putmem(dest, src, pe, flag=None, signal=None)`
- `SignalWait(flag, signal)`

Using `Putmem`, memory is issued to the `dest` array at a remote GPU `pe` from a given local `src` array. As the programming model requires both arrays to be allocated on the symmetric heap, we automatically mark data containers touched by this communication node and resolve to the appropriate memory allocator in the code generation phase. By default, the `Putmem` call used in this way expands to the `shmem_putmem` variant of the upstream library API, specialized for transferring a contiguous chunk of memory.

The `SignalWait` function provides granular synchronization among peer devices through signaling operations on flags in symmetrical memory. The signals are issued by the `Putmem` function through the optional `flag` and `signal` arguments, which atomically communicate and set a given flag to a signal, to be later waited on by `SignalWait`. Using these flags change the default expansion of `Putmem` to `shmem_putmem_signal_wait` that provides said atomicity.

2) Relaxed ordering of operations: In order to achieve a greater level of asynchrony, we employ a relaxed memory model that prioritizes non-blocking memory operations and does not issue synchronization unless explicitly requested. The compiler issues subsequent remote memory writes concurrently, and does not consider possible read/write conflicts of the data involved in `Putmem` calls. Instead, it internally keeps a mapping of signals to array indices updated with `Putmem`, and only considers a write conflict when a `SignalWait` is called on the flag in question. This kind of relaxed ordering of operations allows scheduling remote memory writes in a fire-and-forget fashion.

However, we employ a slightly stricter memory contention policy on local data - `Putmem` calls by default use the blocking `shmem_putmem` API that returns after the source array is copied out and put in transit, in case it is written to by subsequent operations. We relax this requirement when there is no such read/write contention, or an explicit synchro-

nization is present, and switch to the non-blocking variant `shmem_*_nbi` that returns immediately.

3) Scheduling NVSHMEM calls to blocks: Both `NVSHMEM` and `ROC_SHMEM` provide extensions to the OpenSHMEM standard that utilize multiple threads cooperatively for remote memory operations, either at the warp or the block levels, working in tandem with the building block of persistent execution. These operations see increased performance as they make better use of parallel execution in their respective scope [32]. `Putmem` calls expand to block-specialized versions in all cases and are scheduled concurrently. We discuss our scheduling strategy in greater detail in the next section.

4) Strided data transfers: We keep a consistent interface to `Putmem` regardless of the shape of the input, allowing it to accept both contiguous and strided data. Array accesses can be expressed in high-level Numpy syntax, from which the compiler determines the stride and shape of memory. OpenSHMEM, and consequently `NVSHMEM` and `ROC_SHMEM`, provide facilities for strided data movement through the `shmem_iput` API - the compiler changes the expansion of `Putmem` to this API when strided access is detected.

However, the atomic signaling semantics mentioned previously are absent from the strided API: the compiler cannot stage a memory update and a flag set atomically when a signal argument is present. To ameliorate this while conforming to the same `Putmem` interface, we issue additional synchronization in the generated code through the `nvshmemx_signal_op` call. It should be noted that signaling in this way requires stricter memory ordering, as the sending side needs to ensure the completion of the outstanding `iput` data movement before setting the signal. We issue an additional `nvshmem_quiet` to resolve this. As a result, strided data movement has a bigger cost.

C. Concurrent Thread Block Allocation

Many applications contain steps that that be run concurrently - communication calls and memory movement operations are often scheduled in this way to hide their latency. The `NVSHMEM` methods described above in particular benefit from concurrent execution.

As our compiler performs persistent fusion of all steps, we make use of thread block(TB) specialization introduced previously to achieve concurrence. Our implementation discussed below has two parts: we first implement a scheduling strategy for concurrent elements in a given program done automatically by the compiler. We then introduce a Python API to enable users to structure their code with subroutines that are scheduled concurrently.

1) Automatic Thread Block Scheduling: DaCe's SDFG IR provides us with facilities to determine concurrent operations within a given state. Using this, our proposed code generation strategy iteratively schedules each concurrent operation to a subsequent thread or a thread block, omitting synchronization.

```
for t in range(1, TSTEPS):
   nvshmem.Putmem(A[-1, 1:-1], A[1, 1:-1], nn, flag[0], t)
   nvshmem.Putmem(A[0, 1:-1], A[-2, 1:-1], ns, flag[1], t)
   A_pack_w = A[1:-1, 1]
   nvshmem.Putmem(A[1:-1, -1], A_pack_w, nw, flag[2], t)

   A_pack_e = A[1:-1, -2]
   nvshmem.Putmem(A[1:-1, -1], A_pack_e, ne, flag[3], t)

   # ...
```

Listing 4: Concurrent NVSHMEM operations

Consider a sample program performing remote memory update among 4 neighbors in Listing 4. Following the discussion in the previous section, each call to `Putmem` can be concurrently scheduled, as they each update a separate flag.

```
for (auto t = 1; t < TSTEPS; t = t + 1) {
   if (grid.block_rank() == 0) {
      nvshmemx_putmem_signal_nbi_block(&A[1], ...);
   }

   // ...

   if (grid.block_rank() == 3) {
      memcpy_block(A_pack_W, &A[...];
      block.sync();
      nvshmemx_putmem_signal_nbi_block(A_pack_w, ...)
   }
}
```

Listing 5: Concurrent allocation in CUDA

Listing 5 demonstrates sample generated code with concurrent blocks. We note that the memory operations and other dependencies preceding `Putmem` calls can be scheduled within the same thread block. We implement and make use of block-specialized copy routines to better utilize the resources for those cases.

2) Python API: We implement abstractions on the Python side to allow the programmer to schedule code to thread blocks manually. We introduce the following building blocks to denote concurrent code:

1) `Grid` function decorated with `@mapgridscope`. This function denotes the beginning of a scope that spans the entire kernel grid.
2) `Block` functions contained within the scope. These functions are the concurrent elements that are scheduled to one or more thread blocks. The functions do not overlap.

Together, the two constructs allow the programmer to manage block distribution to varying degrees. The main difference between the automatic concurrency described above and this approach is that concurrent subroutines can be scheduled to more than one thread block, allowing a more granular distribution of resources. We define a simple heuristic for determining this based on the number of elements accessed and then outline two examples where the heuristic is used.

$$num\_TB\_subroutine = ceil(\frac{num\_elems\_sub}{total\_num\_elems} \times TB\_count)$$

1. Ranges defined statically. Consider a basic program performing memory copy utilizing TB specialization with two concurrent regions:

```
def prog(A: np.float64[N], A: np.float64[N]):
   @mapgridscope
   def grid():
      def block1(i: _[0:N / 3]):
         B[i] = A[i]
      def block2(i: _[N / 3:]):
         B[i] = A[i]
```

Listing 6: TB specialization with explicit 1/3 + 2/3 split

Both subroutines contain explicit ranges (`[0:N/3]`, `[N/3:]`), coinciding to $\frac{1}{3}$ and $\frac{2}{3}$ of available thread blocks, respectively.

2. Ranges inferred from code. Consider an explicitly overlapped 1D Stencil code with communication:

```
def prog(B: np.float64[N]):
   for t in range(STEPS):
      @mapgridscope
      def grid():
         def block_first():
            B[1] = compute(...)
            communicate(B[1])
         def block_middle():
            B[2:-2] = compute(...)
         def block_last():
            B[-2] = compute(...)
            communicate(B[2])
```

Listing 7: TB specialization with implicit split where first and last thread blocks are spared for communication, and the remaining thread blocks are reserved for computation

Splits cannot be taken from the subroutines directly in this example, as the proportions of elements accessed in each block in regards to one another are unknown at compile time. We can instead perform a best-effort split and clip `block_first` and `block_last` to one thread block each as they both write to and communicate one element, and allocate the remainder of the grid to `block_middle`.

D. Communication Overlap: An example with Stencil

The stencil computation involves updating the value of each element on a grid based on the values of its neighboring elements in a fixed pattern (the stencil). These computations are commonly found in scientific applications and image processing. We implement multi-GPU versions of 2D and 3D stencils from the Polybench suite. Being iterative kernels, they benefit greatly from autonomous execution, as shown in previous work [4], thanks to the execution model eliminating many of the host-side calls and synchronization. For each kernel, we decompose the domain in a tiled fashion that maximizes communication, meaning each device has 4 and 6 neighbors, for 2D and 3D variants, respectively. Due to their shapes, the kernels require both contiguous and strided data movement.

```
def jacobi_2d(TSTEPS, A, B):
   flags = np.full(8, -1, dtype=np.uint64)

   for t in range(1, TSTEPS):
      nvshmem.Putmem(A[-1, 1:-1], A[1, 1:-1], nn,
           flags[0], t)
      nvshmem.Putmem(A[0, 1:-1], A[-2, 1:-1], ns,
           flags[1], t)
      nvshmem.Putmem(A[1:-1, -1], A[1:-1, 1], nw,
           flags[2], t)
      nvshmem.Putmem(A[1:-1, 0], A[1:-1, -2], ne,
           flags[3], t)

      nvshmem.SignalWait(flags[0], t)
      nvshmem.SignalWait(flags[1], t)
      nvshmem.SignalWait(flags[2], t)
      nvshmem.SignalWait(flags[3], t)

      B[1:-1, 1:-1] = 0.2 * (A[...])
```

Listing 8: Multi-GPU 2D Jacobi with NVSHMEM

Listing 8 shows an implementation of a 2D Jacobi Stencil. We perform halo exchange with all neighbors using `Putmem` for each iteration followed by `SignalWait` for correctness. As discussed previously, the consecutive `Putmem` operations are concurrently scheduled to thread blocks and synchronization occurs on `SignalWait` before starting the computation. Though the remote writes occur asynchronously, we enforce synchronization with `SignalWait` immediately after scheduling them, limiting the amount of overlap we get. We can improve this by modifying the communication slightly.

```
def jacobi_2d(TSTEPS, A, B):
   flags = np.full(8, -1, dtype=np.uint64)

   # Communication buffer
   A_prev = np.empty((4, len(A)), dtype=A.dtype)
   A_prev[0] = A[-1, 1:-1]
   # ...

   for t in range(1, TSTEPS):
      nvshmem.Putmem(A_prev[0], A[1, 1:-1], nn, flags[0],
           t)
      nvshmem.Putmem(A_prev[1], A[-2, 1:-1], ns,
           flags[1], t)
      nvshmem.Putmem(A_prev[2], A[1:-1, 1], nw, flags[2],
           t)
      nvshmem.Putmem(A_prev[3], A[1:-1, -2], ne,
           flags[3], t)

      B[1:-1, 1:-1] = 0.2 * (A[...])

      nvshmem.SignalWait(flags[0], t)
      nvshmem.SignalWait(flags[1], t)
      nvshmem.SignalWait(flags[2], t)
      nvshmem.SignalWait(flags[3], t)

      A[-1, 1:-1] = A_prev[0]
      A[0, 1:-1] = A_prev[1]
      A[1:-1, -1] = A_prev[2]
      A[1:-1, 0] = A_prev[3]
```

Listing 9: Multi-GPU 2d Jacobi with NVSHMEM with better communication and computation overlap

Listing 9 shows an alternative implementation that delays calls to `SignalWait` until the end of the iteration. We introduce an additional communication buffer (`A_prev`) to avoid write contentions on the array `A` and allow the scheduler to issue non-blocking remote memory calls.

IV. Evaluation

This section evaluates the performance of our compiler. We aim to evaluate the general applicability of persistent scheduling to a varied range of use cases and identify applications where even single GPU instances benefit. In addition, we present results for multi-GPU execution of stencil computation. The experiments are conducted on a node with 8x NVIDIA Ampere 100 GPUs, with an AMD 7763 64-core processor.

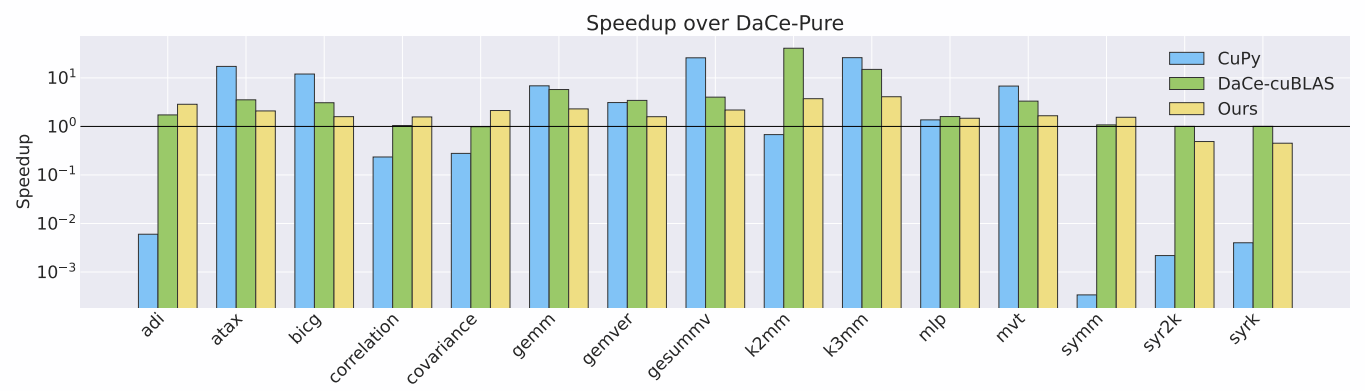

Fig. 3: Relative performance of our compiler compared to the baseline DaCe for applications containing cuBLAS operations using small domains in NPBench.

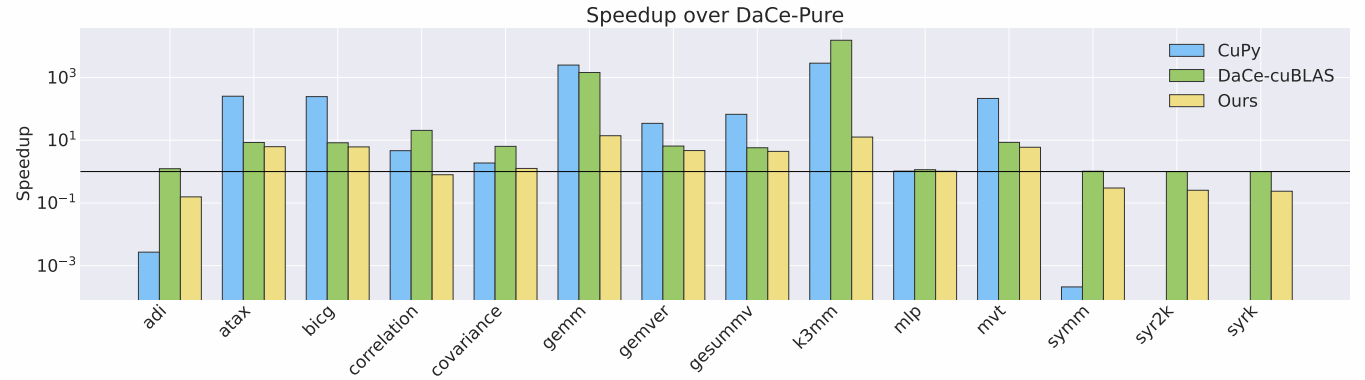

Fig. 4: Relative performance of our compiler compared to the baseline DaCe for applications containing cuBLAS operations using large domains in NPBench.

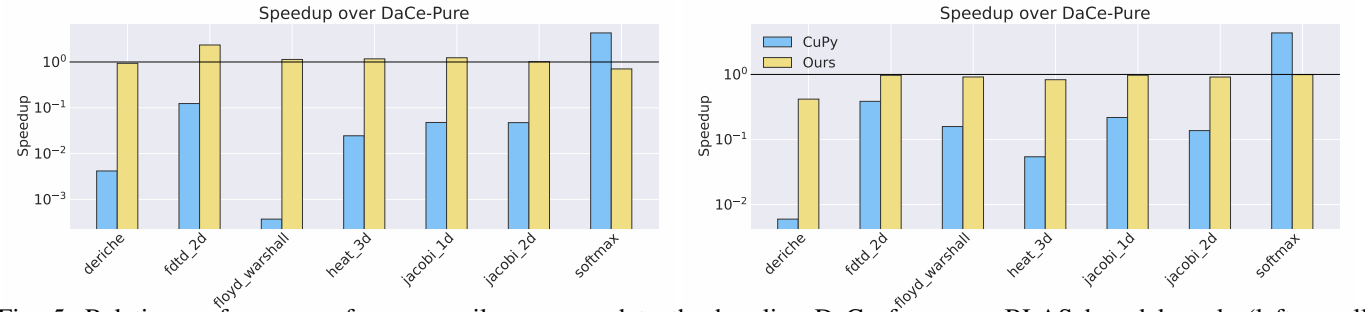

Fig. 5: Relative performance of our compiler compared to the baseline DaCe for non-cuBLAS based kernels (left: small domains, right: large domains in NPBench)

A. Single-GPU Experiments

This portion of our experiments demonstrates our persistent scheduling on a variety of kernels running on a single GPU. We adapt our experiments from NPBench [20], [34] evaluating the performance of Python frameworks on high-level code from several application domains. The experiments reported include several HPC applications such as the Polybench suite that are implemented in high-level Python with Numpy syntax and expressions.

Figures 3, 4 and 5 show the performance of our persistently scheduled kernels compared to existing `DaCe` baselines for small and large domains. We additionally include `CuPy` runs for a more comprehensive comparison. The program versions we report are as follows:

- **DaCe-Pure**: `DaCe` code generation in pure `CUDA`, which serves as the main baseline. This version generates pure `CUDA` code with no libraries, including for BLAS and GEMM.
- **DaCe-cuBLAS**: `DaCe` with `cuBLAS` enabled. This version utilizes the optimized `cuBLAS` library for BLAS

and GEMM (general matrix multiply) operations when possible.
- **Ours**: Our implementation. Since we cannot utilize `cuBLAS` in persistent kernels, this version is derived from and is most similar to `DaCe-Pure`.

For all versions we allow the `DaCe` compiler to perform automatic optimizations on its internal representation before generating and pre-compiling the code. The speedups reported here for all versions are based on the average kernel running time, excluding parsing, optimization, compilation time, memory movement operations, and warmup runs.

We categorize the applications into the following groups:

- **cuBLAS**: Kernels that consist entirely of matrix multiplication and linear algebra operations such as `atax`, `bicg`, `gemm`, `gemver`, `gesummv`, `k2mm`, `k3mm`, `mvt`. Since these kernels can utilize `cuBLAS` for almost the entirety of their execution, we expect lower performance in persistent versions.
- **Iterative**: Kernels that have a top-level time-loop with the bulk of the computation repeated within. This category includes the stencils: `adi`, `fdtd_2d`, `heat_3d`, `jacobi_1d`, and `jacobi_2d`. We expect that our compiler will match or exceed the performance of the baseline for the applications in this category.
- **Others**: Kernels that do not fit into the categories above, and include a mix of BLAS, GEMM, and other operations with no iterative components. Though these kernels can also utilize `cuBLAS`, we expect less of an impact than pure GEMM kernels, as they may benefit from persistent fusion of their steps.

Figure 3 shows the relative performance of the versions discussed above as speedup over `DaCe-Pure` on small domains for kernels containing cuBLAS operations. Persistent kernels perform consistently worse compared to the `DaCe-cuBLAS` and `CuPy` baselines. We observe, however, that persistent scheduling on top of the `DaCe-Pure` codegen nets significant improvements in performance, approaching `cuBLAS` in some cases. Switching to larger domains in Figure 4, we observe an overall degradation in performance in both `DaCe-Pure` and persistent scheduling over `DaCe-cuBLAS`. Overall, our persistent scheduling gains a geometric mean speedup of $1.49x$ on small domains and $1.36x$ on large domains, over `DaCe-Pure`, though `DaCe-cuBLAS` outperforms both in all cases.

In Figure 5, among the iterative kernels at small domains, we observe equal or better performance across the board with noticeable speedups over the baseline `Dae-Pure` codegen in `adi`, `fdtd_2d` and `jacobi_1d`. `heat_3d`, and `jacobi_2d` match the performance of baselines within 10.3% and 2.8%, respectively. `jacobi_1d` gains the most improvement overall, performing 17.05% better than the baseline.

As for the large domain runs, iterative persistent kernels match the performance of the baselines within 5.85%, 2.5% and 1.1% for `fdtd_2d` `jacobi_1d` and `jacobi_2d`, respectively. `heat_3d` takes a noticeable hit, performing 16.9% worse compared to the baseline. `adi` displays the worst results, performing more than $7x$ worse than the pure baseline.

B. Multi-GPU Experiments

We conduct scaling experiments of the multi-GPU stencil implementations discussed in Section III-D. We compare our implementation to baseline `DaCe` with MPI communication as well as `cuNumeric`. Figure 6 shows strong scaling results on 8 GPUs in a single node. For each version, we begin with a sufficiently large domain size and divide it in half for each step while doubling the number of GPUs. The reported metrics are mean execution time for 5 runs.

1) 2D Jacobi: 2D Jacobi kernel requires a large amount of data to be communicated among four neighbors. Furthermore, since we partition the domain as a grid in both `DaCe` and our version, two of these neighbors require strided data transfers.

We observe the effects of our scheduling strategy and GPU-initiated communication in this application, gaining significant speedups over both baselines. Baseline `DaCe` becomes dominated by communication immediately, likely due to the larger number of neighbors and host-side packing and unpacking done for strided MPI routines. `cuNumeric` also has significantly worse performance from the start, though it initially exhibits adequate scaling, experiencing significant degradation after 4 GPUs. Our version performs consistently better than both baselines and achieves a speedup of $23.1x$ over `cuNumeric` at 8 GPUs.

2) 3D Jacobi: We change the communication scheme for this kernel and split the domain across the contiguous dimension, without needing strided data movement. This results in each rank communicating a chunk of memory with two neighbors. Though both baselines exhibit adequate scaling initially, `cuNumeric`, similar to Jacobi 2D, plateaus after 4 GPUs. Baseline `DaCe` continues to scale, though not reaching our persistent implementation.

Overall, we observe a speedup of $6.1x$ and $30.8x$ over `DaCe` and `cuNumeric`, respectively, on 8 GPUs. Our version achieves a scaling efficiency of 98% at 8 GPUs compared to a single GPU run with communication disabled.

V. Related Work

Several other works have explored the components we used to build a CPU-free compiler. Persistent kernels were first introduced in the work by Gupta et al. [22] who formalized a Persistent Threads programming model and identified its benefits for irregular applications. Later work by Chu et al. [35] adapt the concept to a GPU key-value store, while other works employ it for iterative workloads [4], [23]. Thread Block specialization has seen use in several frameworks such as WhippleTree [3], Groute [36], Juggler [25], Atos [5], and more.

GPU-Initiated communication has seen increasing adoption in recent literature thanks to the introduction of NVSHMEM [5]–[9], [32]. Several frameworks and libraries have added

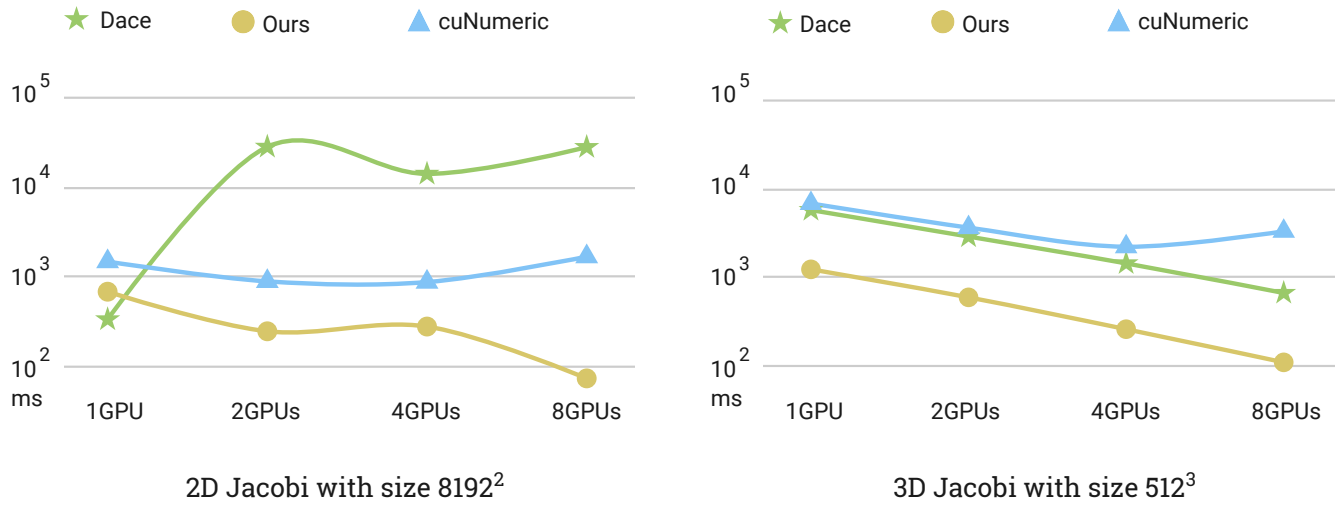

Fig. 6: Strong scaling results for multi-GPU Jacobi stencil in 2D and 3D (Time in ms, lower is better)

support for it including LBANN [37], which uses its one-sided put operation to gain significant benefit on spatial-parallel convolution [15]. Kokkos includes NVSHMEM as a communication backend in their Remote Spaces API, which has been shown to net significant performance improvement over MPI on a Conjugate Gradient solver [15]. QUDA is another library that ships with NVSHMEM support, where it has been shown to achieve better scaling performance over MPI on Wilson Dslash kernels [38].

Much work has gone into high-level frameworks and libraries in Python that facilitate writing full-scale distributed applications. DaCe is a multi-target framework that generates optimized parallel code from Numpy-style Python code for CPUs, FPGAs, and AMD and NVIDIA GPUs [18]. The framework has support for multi-GPU communication through user-defined MPI routines, available as Python abstractions. CuPy is another accelerated Numpy-compatible framework that compiles into CUDA source code. The library can be used in distributed settings with its NCCL support or together with mpi4py [39], which provides Python bindings to MPI.

cuNumeric provides a similar interface and builds on top of the Legate framework [40], which provides task-based parallelism and automatic data partitioning for multi-GPU systems. It orchestrates communication automatically and uses NCCL and UCX as the backend. Dask works on top of CuPy and other compatible libraries and builds a task scheduler that automatically distributes computation across devices and nodes [19]. The library has support for several communication backends, including MPI, NCCL, and UCX. Finally, frameworks such as Ray [41], PyTorch [42], and Tensorflow [43] specialize on machine learning applications and provide distributed computing using NCCL. Other frameworks such as Charm4py [44] and PyKokkos [45] offload to existing libraries and utilize their facilities for accelerated and distributed computing. Charm4py provides bindings to the Charm++ parallel runtime [46] which utilizes MPI for communication, while PyKokkos translates Python code to C++ and Kokkos [16].

VI. Conclusion

In this paper, we designed and developed a code generator for a CPU-free autonomous execution model, which departs from the traditional host-based GPU execution. Our contributions include providing a Python-based API that enables writing CPU-free code with GPU-centric constructs and GPU-initiated communication calls. We implemented a code-generation backend that creates persistent kernels and efficiently schedules threads and thread blocks. Additionally, we developed an autonomous communication layer using `NVSHMEM` and `ROC_SHMEM`, which enables GPU-initiated communication directly in Python code. Our performance study, conducted on 22 kernels from the NPBench suite on a single GPU, demonstrated the effectiveness of our implementation compared to DaCe and CuPy. We also compared the performance of 2D and 3D multi-GPU stencil computations against multi-GPU DaCe and cuNumeric, highlighting our approach's advantages. Our future work will include more application studies and scalability studies on multi-node systems.

Acknowledgment

This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 949587).

References

[1] S. Tzeng, A. Patney, and J. D. Owens, "Task management for irregular-parallel workloads on the gpu," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10. Goslar, DEU: Eurographics Association, 2010, p. 29–37.

[2] M. Bauer, S. Treichler, and A. Aiken, "Singe: leveraging warp specialization for high performance on gpus," *SIGPLAN Not.*, vol. 49, no. 8, p. 119–130, feb 2014. [Online]. Available: https://doi.org/10.1145/2692916.2555258

[3] M. Steinberger, M. Kenzel, P. Boechat, B. Kerbl, M. Dokter, and D. Schmalstieg, "Whippletree: Task-based scheduling of dynamic workloads on the gpu," *ACM Trans. Graph.*, vol. 33, no. 6, nov 2014. [Online]. Available: https://doi.org/10.1145/2661229.2661250

[4] I. Ismayilov, J. Baydamirli, D. Sağbili, M. Wahib, and D. Unat, "Multi-gpu communication schemes for iterative solvers: When cpus are not in charge," in *Proceedings of the 37th International Conference on Supercomputing*, ser. ICS '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 192–202. [Online]. Available: https://doi.org/10.1145/3577193.3593713

[5] Y. Chen, B. Brock, S. Porumbescu, A. Buluç, K. Yelick, and J. D. Owens, "Scalable irregular parallelism with gpus: Getting cpus out of the way," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22. New York, NY, USA: Institute for Electrical and Electronics Engineers, 2022.

[6] S. Potluri, D. Rossetti, D. Becker, D. Poole, M. Gorentla Venkata, O. Hernandez, P. Shamis, M. G. Lopez, M. Baker, and W. Poole, "Exploring openshmem model to program gpu-based extreme-scale systems," in *Revised Selected Papers of the Second Workshop on OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies - Volume 9397*, ser. OpenSHMEM 2015. Berlin, Heidelberg: Springer-Verlag, 2015, p. 18–35. [Online]. Available: https://doi.org/10.1007/978-3-319-26428-8_2

[7] S. Potluri, A. Goswami, D. Rossetti, C. Newburn, M. G. Venkata, and N. Imam, "Gpu-centric communication on nvidia gpu clusters with infiniband: A case study with openshmem," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. New York, NY, USA: Institute for Electrical and Electronics Engineers, 2017, pp. 253–262.

[8] S. Potluri, A. Goswami, M. G. Venkata, and N. Imam, "Efficient breadth first search on multi-gpu systems using gpu-centric openshmem," in *OpenSHMEM and Related Technologies. Big Compute and Big Data Convergence*, M. Gorentla Venkata, N. Imam, and S. Pophale, Eds. Cham: Springer International Publishing, 2018, pp. 82–96.

[9] C.-H. Hsu, N. Imam, A. Langer, S. Potluri, and C. J. Newburn, "An initial assessment of nvshmem for high performance computing," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. New York, NY, USA: Institute for Electrical and Electronics Engineers, 2020, pp. 1–10.

[10] J. Choi, D. F. Richards, and L. V. Kale, "Charming: A scalable gpu-resident runtime system," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 261–262. [Online]. Available: https://doi.org/10.1145/3431379.3464454

[11] M. LeBeane, K. Hamidouche, B. Benton, M. Breternitz, S. K. Reinhardt, and L. K. John, "Gpu triggered networking for intra-kernel communications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126950

[12] K. Hamidouche and M. LeBeane, "Gpu-initiated openshmem: correct and efficient intra-kernel networking for dgpus," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 336–347. [Online]. Available: https://doi.org/10.1145/3332466.3374544

[13] J. Ciesko, "Distributed memory programming and multi-gpu support with kokkos." 11 2020. [Online]. Available: https://www.osti.gov/biblio/1829951

[14] J. Zhang, J. Brown, S. Balay, J. Faibussowitsch, M. Knepley, O. Marin, R. T. Mills, T. Munson, B. F. Smith, and S. Zampini, "The petscsf scalable communication layer," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, 5 2021. [Online]. Available: https://www.osti.gov/biblio/1837203

[15] B. V. E. Naoya Maruyama, J. Ciesko, J. Wilke, C. Trott, C.-H. Hsu, N. Imam, J. Dinan, A. Langer, C. Newburn, and S. Potluri, "Scaling scientific computing with nvshmem," 2020. [Online]. Available: https://developer.nvidia.com/blog/scaling-scientific-computing-with-nvshmem/

[16] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *2013 Extreme Scaling Workshop (xsw 2013)*, 2013, pp. 18–24.

[17] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "Raja: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81.

[18] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, 2019.

[19] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th python in science conference*, no. 130-136. Citeseer, 2015.

[20] A. N. Ziogas, T. Ben-Nun, T. Schneider, and T. Hoefler, "Npbench: a benchmarking suite for high-performance numpy," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 63–74. [Online]. Available: https://doi.org/10.1145/3447818.3460360

[21] Y. Chen, B. Brock, S. Porumbescu, A. Buluc, K. Yelick, and J. Owens, "Atos: A task-parallel gpu scheduler for graph analytics," in *Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3545008.3545056

[22] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *2012 Innovative Parallel Computing (InPar)*. New York, NY, USA: Institute for Electrical and Electronics Engineers, 2012, pp. 1–14.

[23] L. Zhang, M. Wahib, P. Chen, J. Meng, X. Wang, and S. Matsuoka, "Persistent kernels for iterative memory-bound gpu applications," 2022.

[24] L. Zhang, M. Wahib, H. Zhang, and S. Matsuoka, "A study of single and multi-device synchronization methods in nvidia gpus," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. New York, NY, USA: Institute for Electrical and Electronics Engineers, 2020, pp. 483–493.

[25] M. E. Belviranli, S. Lee, J. S. Vetter, and L. N. Bhuyan, "Juggler: A dependence-aware task-based execution framework for gpus," *SIGPLAN Not.*, vol. 53, no. 1, p. 54–67, feb 2018. [Online]. Available: https://doi.org/10.1145/3200691.3178492

[26] K. Hamidouche, A. Venkatesh, A. A. Awan, H. Subramoni, C.-H. Chu, and D. K. Panda, "Exploiting gpudirect rdma in designing high performance openshmem for nvidia gpu clusters," in *2015 IEEE International Conference on Cluster Computing*. New York, NY, USA: Institute for Electrical and Electronics Engineers, 2015, pp. 78–87.

[27] D. Foley and J. Danskin, "Ultra-performance pascal gpu and nvlink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[28] H. Wang, S. Potluri, M. Luo, A. Singh, S. Sur, and D. Panda, "Mvapich2gpu: optimized gpu to gpu communication for infiniband clusters," *Computer Science - Research and Development*, vol. 26, pp. 257–266, 06 2011.

[29] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda, "Optimized non-contiguous mpi datatype communication for gpu clusters: Design, implementation and evaluation with mvapich2," in *2011 IEEE International Conference on Cluster Computing*, 2011, pp. 308–316.

[30] OpenMPI, "Open MPI v5.0.x Documentation: ROCm," https://docs.open-mpi.org/en/v5.0.x/tuning-apps/networking/rocm.html, 2023.

[31] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. Van Essen, "Aluminum: An asynchronous, gpu-aware communication library optimized for large-scale training of deep neural networks on hpc systems," in *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, 2018, pp. 1–13.

[32] NVIDIA, "Nvidia openshmem library (nvshmem) documentation," 2022. [Online]. Available: https://docs.nvidia.com/nvshmem/api/

[33] V. G. Yuan Lin, "Using cuda warp-level primitives," 2018. [Online]. Available: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/

[34] A. N. Ziogas, T. Schneider, T. Ben-Nun, A. Calotoiu, T. De Matteis, J. de Fine Licht, L. Lavarini, and T. Hoefler, "Productivity, portability, performance: data-centric python," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476176

[35] C.-H. Chu, S. Potluri, A. Goswami, M. Gorentla Venkata, N. Imam, and C. J. Newburn, "Designing high-performance in-memory key-value operations with persistent gpu kernels and openshmem," in *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity*, S. Pophale, N. Imam, F. Aderholdt, and M. Gorentla Venkata, Eds. Cham: Springer International Publishing, 2019, pp. 148–164.

[36] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-gpu programming model for irregular computations," *SIGPLAN Not.*, vol. 52, no. 8, p. 235–248, jan 2017. [Online]. Available: https://doi.org/10.1145/3155284.3018756

[37] B. Van Essen, H. Kim, R. Pearce, K. Boakye, and B. Chen, "Lbann: livermore big artificial neural network hpc toolkit," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, ser. MLHPC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2834892.2834897

[38] M. Wagner, "Scaling lattice qcd on modern gpu systems," 2019. [Online]. Available: https://agenda.infn.it/event/17130/contributions/106939/attachments/69343/86116/Wagner_QUDA_SMFT2019.pdf

[39] L. Dalcin and Y.-L. L. Fang, "mpi4py: Status Update After 12 Years of Development," *Computing in Science & Engineering*, vol. 23, no. 4, pp. 47–54, 2021.

[40] M. Bauer and M. Garland, "Legate numpy: Accelerated and distributed array computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356175

[41] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging ai applications," 2018.

[42] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019.

[43] 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. Mané, 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. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[44] J. J. Galvez, K. Senthil, and L. Kale, "Charmpy: A python parallel programming model," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 423–433.

[45] N. Al Awar, N. Mehta, S. Zhu, G. Biros, and M. Gligoric, "Pykokkos: Performance portable kernels in python," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022, pp. 164–167.

[46] L. V. Kale, "The Charm++ Parallel Programming System." [Online]. Available: https://github.com/charmplusplus/charm