An Adaptive Distributed Stencil Abstraction for GPUs

Aditya Bhosale
University of Illinois at Urbana-Champaign
Urbana, IL, USA
adityapb@illinois.edu

Laxmikant Kale
University of Illinois at Urbana-Champaign
Urbana, IL, USA
kale@illinois.edu

Abstract—The scientific computing ecosystem in Python is largely confined to single-node parallelism, creating a gap between high-level prototyping in NumPy and high-performance execution on modern supercomputers. The increasing prevalence of hardware accelerators and the need for energy efficiency have made resource adaptivity a critical requirement, yet traditional HPC abstractions remain rigid. To address these challenges, we present an adaptive, distributed abstraction for stencil computations on multi-node GPUs. This abstraction is built using CharmTyles, a framework based on the adaptive Charm++ runtime, and features a familiar NumPy-like syntax to minimize the porting effort from prototype to production code. We demonstrate that our abstraction achieves significant performance improvements over both a specialized, high-performance stencil DSL and a generalized NumPy replacement. Furthermore, we showcase its resource elasticity by dynamically rescaling a running application across a different number of nodes and present a performance analysis of the associated overheads.

Index Terms—stencil, GPU, elasticity

I. INTRODUCTION

Python has emerged as a language of choice for scientific computing due to its large ecosystem of composable libraries built upon NumPy's common array representation [14]. In particular, NumPy's slicing notation is widely used for implementing numerical methods, such as finite difference and finite volume, over structured grids. This pattern of stencil computations has a wide range of applications, including computational fluid dynamics, seismic imaging, electromagnetism, and image processing. While NumPy exhibits good single-threaded performance by internally calling highly efficient BLAS functions and supports multi-threaded execution for certain operations, its parallelism is limited to a single node.

Over the last few decades, parallel programming has seen a steady shift from shared-memory architectures to distributed-memory systems and now to hardware accelerators like GPUs and TPUs. Each new hardware architecture introduces its own programming model, optimization challenges, and performance constraints. This changing landscape of parallel programming has made it difficult for domain scientists to effectively utilize available resources. Domain-specific languages (DSLs) are a powerful tool to separate domain expertise from the technical knowledge required to extract optimal performance from different hardware architectures. To that

end, several Python-based DSLs have been developed for high-performance stencil computations [16], [17], [24].

A common workflow in scientific computing is to first build a prototype using NumPy to evaluate an algorithm on a smaller scale and then develop a high-performance implementation. Unfortunately, since these high-level DSLs often differ significantly in syntax from NumPy, the effort required to port the implementation is substantial. Several NumPy drop-in replacements have been developed to tackle this issue [3], [23]; however, these replacements often compromise on domain-specific performance optimization in the pursuit of generality.

Due to the rise in popularity of AI, the demand for accelerators has increased exponentially over the past few years. This sudden rise in demand has also strained the energy requirements for datacenters and supercomputers, making it more important than ever to use these resources efficiently. Moreover, as issues like hardware failures and power constraints become more pertinent in modern supercomputing systems, it is crucial to introduce adaptivity and resource awareness into job schedulers and parallel programming models. The widespread adoption of cloud platforms for HPC workloads has underscored the importance of resilience, fault-tolerance, and resource elasticity in parallel programming models and abstractions to effectively use these volatile cloud resources.

Traditionally, parallel programming models like MPI have been rigid. As a result, job schedulers like SLURM have not supported dynamic allocation and adaptivity. In recent years, significant work has been done to develop alternative, adaptive parallel programming models such as Charm++ [13], and to extend MPI to incorporate elasticity [11], [15]. Similarly, there has been work on introducing adaptivity to traditional job schedulers like SLURM [7] and exploring resource elasticity in the HPC-Cloud converged computing paradigm [4], [6], [18], [19]. While several popular libraries for distributed machine learning have adapted to this changing trend by introducing elasticity into their abstractions [12], traditional HPC libraries and DSLs have largely continued to be rigid.

CharmTyles is a framework based on the Charm++ programming model for writing DSLs with a Python frontend and a parallel Charm++ backend [5]. In this paper, we extend the resource elasticity in Charm++ and CharmTyles to support GPUs. We present an adaptive, distributed abstraction for stencil computations on multi-node GPUs built using CharmTyles

with a NumPy-like syntax. We compare the performance of our abstraction to a specialized stencil DSL and a generalized NumPy replacement, showing a significant performance improvement over both. We also demonstrate the adaptivity of our abstraction by dynamically rescaling an application to a different number of nodes, and we measure the performance overheads associated with this rescaling.

II. BACKGROUND

A. Charm++

Charm++ is an asynchronous message-driven parallel programming model [1] where computation is expressed in terms of objects known as *chares*. The Charm++ runtime maps these chares to Processing Elements (PEs) and manages communication between them through remote entry method invocations. A key feature of Charm++ is that its chares are migratable, allowing the runtime system to move them between PEs to perform dynamic load balancing and support resource elasticity.

1) GPU support: GPU support in Charm++ is implemented using the Hybrid API (HAPI) [8]. HAPI provides three main functionalities:

a) Mapping PEs to GPUs: Modern supercomputers have many CPU cores but few GPUs per node, so several PEs must share a single GPU. In Charm++, HAPI manages this PE-to-GPU mapping using built-in or custom, user-defined functions.

b) Asynchronous detection of kernel completion: Charm++ programs often use a continuation-passing style. Due to its message-driven model, an entry method performs local computation and then invokes a remote method to continue. This creates a dependency between the local computation and the subsequent remote invocation.

If the local computation is a CUDA kernel, enforcing this dependency would require a blocking call, which is suboptimal as it can delay other messages in the PE's scheduler. HAPI provides a callback to resolve this: the kernel call is made asynchronous, and a callback containing the subsequent message is registered. The Charm++ runtime delivers this callback only after the kernel is complete.

c) Direct GPU messaging: HAPI supports sending GPU buffers as entry method arguments using the Charm++ zero-copy API. Communication of GPU buffers is performed using GPUDirect RDMA when Charm++ is built with the UCX communication layer [9]. For other communication layers, intra-node GPU transfers are done using CUDA IPC and CUDA memcpy, while inter-node GPU communication uses host-staging.

2) Shrink/Expand support: The migratability of chares enables an application to dynamically scale its number of PEs up or down at runtime. When scaling down (shrinking), chares located on the PEs designated for removal are migrated to the remaining PEs. Conversely, when scaling up (expanding), chares from existing PEs are moved to the newly added PEs.

To manage the internal data structure modifications required by rescaling, the application is checkpointed locally and restarted with the new number of PEs. In a shrink operation, chares are first migrated away from the PEs being removed. The application is then checkpointed to shared memory, restarted with fewer PEs, and restored. During an expand operation, the application is first checkpointed and restarted with the new PEs. After the checkpoint data is restored, a load balancing procedure is executed to migrate chares onto the new PEs.

A signal for an application to rescale can be sent from an external program using the Converse Client Server (CCS) interface [10]. The rescaling operation is then triggered during the application's subsequent load balancing step.

B. CharmTyles

CharmTyles is a framework for writing DSLs using a client-server architecture with a Python frontend and a Charm++ backend [5]. As shown in Figure 1, the framework consists of a Charm++ server program running on the backend. When running on GPU systems, this program uses PEs that are processes or threads running on the CPU, with each PE mapped to a GPU device. Data on the backend is distributed across these PEs as *tiles*.

The frontend is a Python interpreter where the user expresses computation in a high-level DSL. The frontend encodes the computation into a message that is sent via CCS to the backend, where the Charm++ program executes it.

CharmStencil is a distributed stencil abstraction for CPUs that was previously built using the CharmTyles framework [5]. In this paper, we extend CharmStencil to support execution on GPUs and modify the frontend to more closely resemble NumPy's syntax. Henceforth in this paper, "CharmStencil" will refer to the new GPU implementation of this abstraction developed as part of this work.
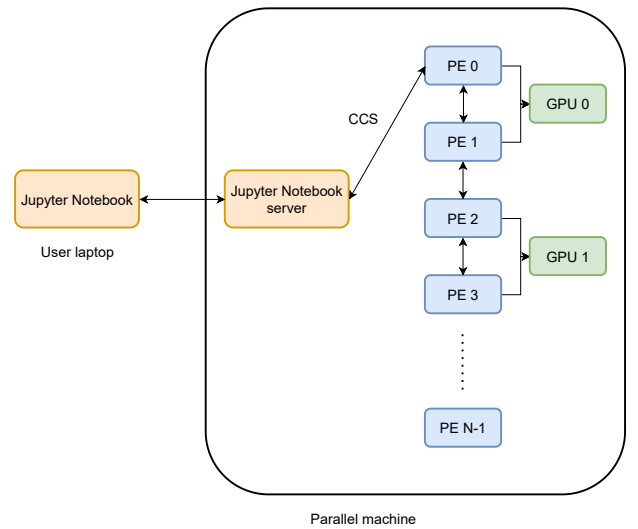

Fig. 1: CharmTyles architecture with GPUs

III. IMPLEMENTATION

A. Overview

We implement our stencil abstraction using the CharmTyles framework. On the frontend, the user writes operations with a NumPy-like syntax. These operations are aggregated and sent to the backend, creating a pipelined execution model that helps hide the overhead introduced by the Python interpreter.

Example 2 shows a 2D Laplace equation solver written using CharmStencil, which has a syntax very similar to a NumPy implementation. A notable difference is that the CharmStencil implementation requires two arrays, u1 and u2, whereas the same application in NumPy could be written with a single array. This design choice avoids the creation of temporary arrays for intermediate results, instead requiring the user to write expressions without data dependencies. The frontend will detect any such dependencies and raise an error.

```
u1 = create_array((16384, 16384))
u2 = create_array((16384, 16384))

u1[0, :] = u1[-1, :] = u1[:, 0] = u1[:, -1] = 1
u2[0, :] = u2[-1, :] = u2[:, 0] = u2[:, -1] = 1

for i in range(10):
    u2[1:-1, 1:-1] = 0.25 * (u1[:-2, 1:-1] +
                             u1[2:, 1:-1] +
                             u1[1:-1, :-2] +
                             u1[1:-1, 2:])
    u1, u2 = u2, u1
```

Fig. 2: Example of a 2D Laplace solver using CharmStencil

B. Frontend

1) Execution graph: The CharmStencil frontend captures array operations as a parameterized Abstract Syntax Tree (AST). An AST is generated to represent the right-hand side of any expression that assigns to an array slice, and each AST stores the input and output arrays involved in that computation. The Python program is then represented as a Directed Acyclic Graph (DAG), where each node is an AST and edges represent true, anti, or output dependencies between the arrays of connected nodes.

For better frontend performance, dependencies are tracked at the array level rather than the index level. While this approach may add an unnecessary edge between two independent nodes, it guarantees that no required dependency is missed.

For example, Figure 3 shows the AST for the update expression inside the loop and the DAG for the entire program in Example 2. The red nodes in the DAG are the creation nodes for the arrays u1 and u2. Because the AST is parameterized, all ten nodes representing the Jacobi relaxation iterations refer to the same AST, even when the arrays are swapped.

The DAG and its corresponding ASTs are sent to the backend when a sync function is called on the frontend, or when the DAG depth exceeds a user-configurable maximum.

2) Node fusion: In numerical methods, it is common for each grid point to have several fields that are updated at each timestep. In our abstraction, these fields are expressed as
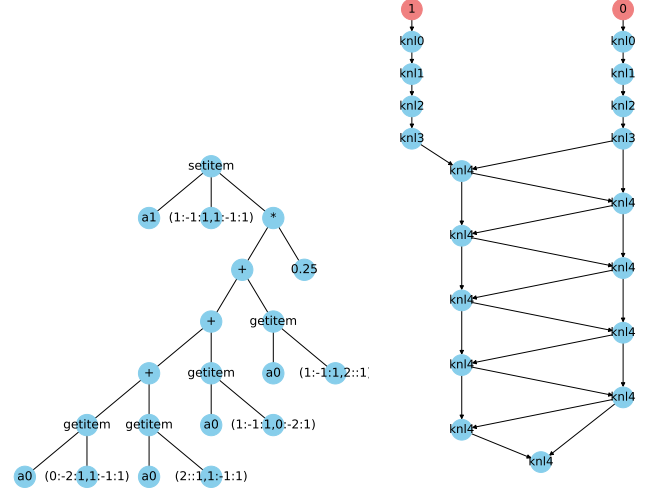
(a) AST generated for the main loop expression (knl4 in the DAG)

(b) DAG generated for the example program 2

Fig. 3: AST and DAG examples for 2D Laplace solver Example 2

separate arrays of the same shape. Because we represent each assignment as a separate AST, cases where multiple fields are updated in a single iteration will result in multiple DAG nodes. These distinct ASTs are then compiled into separate CUDA kernels on the backend, which prevents potential data reuse and incurs additional kernel launch overheads.

To address this issue, we perform a node fusion pass over the DAG before sending it to the backend. We use two criteria to determine if nodes can be fused.

a) Output shape: Fusing nodes with different output shapes would require the generated kernel to have complicated logic to map each thread to the correct output index for each expression. Moreover, if the sizes of the output arrays differ, it could create a load imbalance within the generated kernel. In stencil codes, the main computation loop typically iterates over the same set of grid points for all fields. Therefore, we restrict our node fusion criterion to nodes that have the same output shape.

b) Dependencies: Two nodes can only be fused if they do not have any read-after-write or write-after-read dependencies between them. Since we express these dependencies as edges in our DAG, this criterion means that two nodes can be fused only if there is no path between them. Finding all pairs of nodes in a graph without a path is a computationally expensive problem [20]. However, in typical stencil codes, operations on different fields within an iteration are written consecutively. Thus, instead of checking all pairs of nodes, we use a heuristic that only attempts to fuse consecutive nodes in program order.

C. Backend

1) Code generation: The backend performs an analysis pass on each AST it receives from the frontend. During this
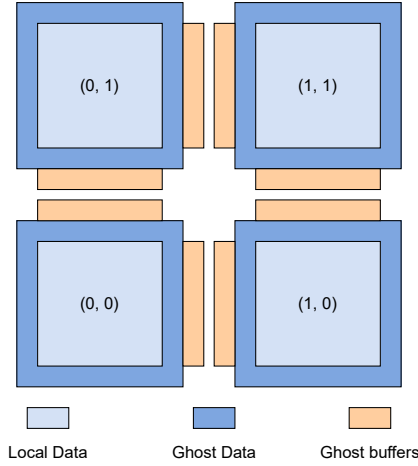
Fig. 4: Data layout for 2×2 chare decomposition for an array with ghost depth 1

pass, it tracks the memory access pattern of each argument to calculate the maximum offset between any input and output index. This information is stored as kernel metadata and is later used to determine the required ghost size for arrays. The backend also tracks data reuse, marking input arguments for use of shared memory if this optimization is enabled. For each kernel, the array arguments that are written to are tracked to assist with optimizing data movement, as discussed later in this section.

Following the analysis, the backend generates a CUDA kernel for each AST. The current code generator implementation assumes that each thread calculates at most one element of an output array slice. Consequently, the launch parameters are configured so that the total number of threads equals the size of the largest output slice. Each process compiles the generated CUDA kernel to PTX separately, after which the kernel is dynamically loaded. This per-process compilation strategy was chosen to support systems without a shared filesystem, such as a cluster of AWS EC2 instances.

2) Data layout: On the backend, array data is tiled and distributed among chares. The total number of chares is determined by the number of PEs the backend is launched with and an over-decomposition factor set by the user at the frontend. All arrays are partitioned equally among all chares.

Figure 4 shows the data layout for a 2×2 chare decomposition. Each chare's data structure has three components: the local data it owns, the ghost data from neighboring chares, and the send and receive buffers used for ghost exchanges. The size of the ghost region is inferred by taking the maximum ghost depth required by any kernel to which the array is passed as an argument.

3) DAG execution: Each chare creates a compute stream and a communication stream. All kernel launches are enqueued in the compute stream, while all memory copies and packing/unpacking kernels for ghost data are enqueued in the com-

munication stream. Each chare also has two CUDA events: a compute event and a communication event. The compute event is recorded in the compute stream after a kernel launch, and the communication event is recorded in the communication stream after the latest memory copy or packing/unpacking kernel. These events are used to synchronize the compute and communication streams on each chare.

After the code for the ASTs is generated, the computation DAG is broadcast to all chares. Each chare executes the nodes in the DAG according to its specified dependencies. The execution of a DAG node occurs in two steps.

First, the ghost data for the input arrays is exchanged asynchronously. Before the packing kernels are called, a wait for the compute event is enqueued in the communication stream. This ensures that previous kernel executions are complete before the ghost data is copied into the send buffer. After the packing and unpacking kernels are called, the communication event is recorded.

Second, when the ghost data for all input arrays is available, a wait for the communication event is enqueued in the compute stream. This ensures that all ghost data has been copied into the send buffer and out of the receive buffer before the main kernel executes. The CUDA kernel is then enqueued in the compute stream. This asynchronous data exchange allows for the overlap of computation and communication between independent nodes in the DAG.

To determine which ghost exchanges are necessary for the execution of a DAG node, the array data structure maintains a generation epoch for both local and ghost data. When a kernel executes, the local generation epoch for the arrays that are written to is incremented. When the ghost data for an array is exchanged, its ghost generation epoch is updated to match the local generation epoch. Before a DAG node is executed, the kernel metadata is checked to identify arrays with a non-zero ghost depth. A ghost exchange is performed only for these arrays and only if their local and ghost generation epochs do not match.

D. Limitations & Future Work

While NumPy's slicing notation is a powerful way to express computations on structured grids, it restricts applications to those where computations can be expressed as operations on array subviews. Consequently, irregular algorithms—such as particle methods and unstructured-grid solvers—cannot be expressed using this abstraction.

Our ghost-region-based remote data transfer scheme places an additional restriction on the application's dependence relation: a grid point i must not depend on a grid point $i \pm k$ where k is greater than the width of a chare, as remote data is only transferred between neighboring chares. Furthermore, our current implementation does not optimize ghost data transfers involving strided views, such as those found in geometric multigrid methods. We plan to address these limitations in future work by implementing a more general remote data transfer mechanism.

IV. RESOURCE ELASTICITY

Resource elasticity requires support from both the job scheduler and the parallel programming model. The job scheduler must be able to dynamically change a job's allocation, while the programming model must be able to adapt to this new allocation. The specifics of job scheduler support are outside the scope of this work. In this paper, we focus on providing elasticity within the programming model.

In Charm++, resource elasticity is supported using its shrink/expand functionality. Previously, this support was limited to CPUs and the `netlrts` communication layer. To enable resource elasticity in our abstraction, we extended the shrink/expand functionality in Charm++ to support GPUs and generalized the implementation to work with any communication layer. Our abstraction uses the UCX layer, which supports GPUDirect communication across devices. Extending shrink/expand support to GPUs involved two major contributions.

1) Migration of chares: Chare migration is an essential part of the shrink/expand process. During a shrink operation, chares on PEs designated for removal are migrated to other PEs. During an expand operation, chares are migrated to new PEs to rebalance the load. The migratability of chares is enabled by the PUP framework in Charm++, which we extended to support the migration of GPU-resident data using GPUDirect RDMA.

2) Checkpoint and restart: During a shrink/expand operation, local chare data is checkpointed to Linux shared memory, the application is restarted with the new number of processes, and the checkpoint data is restored. With GPU-resident data, this approach requires that data persists on the GPU through an application restart. However, the lifetime of data on a GPU is tied to the CUDA context that owns it, and a CUDA context cannot persist across a process restart. One solution is to copy all data from the device to the host and checkpoint it to Linux shared memory. Then, upon restart, the checkpoint data is restored and copied back to the GPU. However, host-to-device transfers are typically expensive due to the low bandwidth of the PCIe bus.

To avoid this costly data movement between the host and device, we launch a separate memory daemon process for every GPU at application startup. When the application is signaled to rescale, the GPU-resident data pointers on all chares are sent to the corresponding memory daemons using CUDA IPC and a named pipe. The memory daemons copy the data into a buffer they own and return an allocation ID to the Charm++ runtime. This allocation ID is written into the shared memory checkpoint, and the application is restarted while the memory daemons remain active. During the application's restore stage, the Charm++ runtime retrieves the device pointer corresponding to the allocation ID and copies the data back into a device buffer allocated by the Charm++ runtime. The corresponding data on the memory daemon is then freed. This scheme replaces expensive device-to-host data movement with more efficient device-to-device copies. Figure 5 shows an end-to-end illustration of shrinking from two PEs on two nodes to
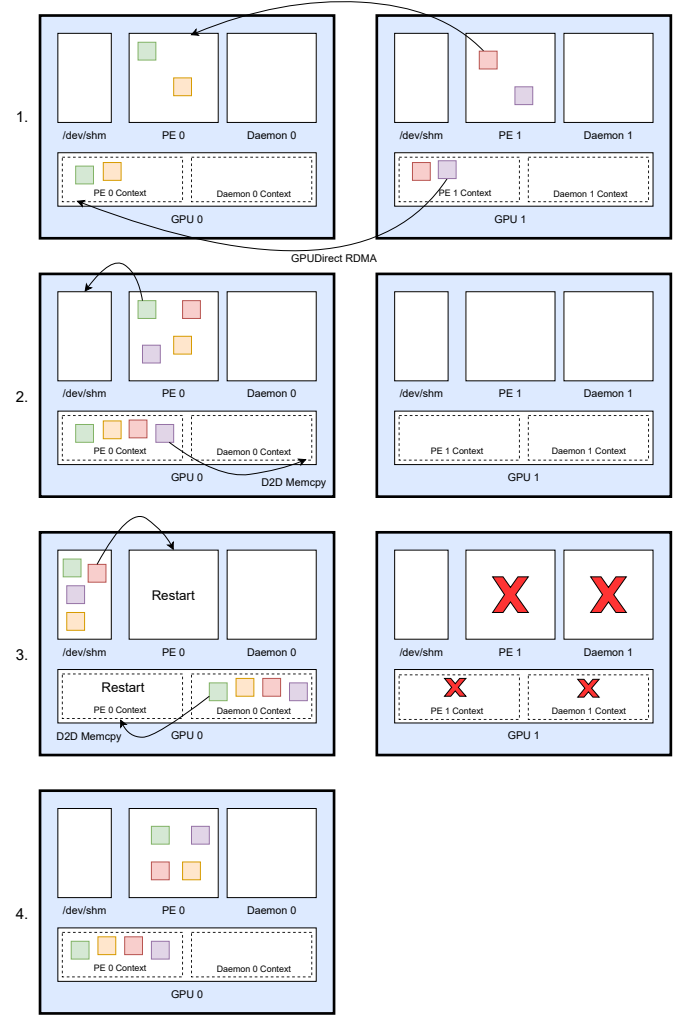

Fig. 5: Shrinking from 2 PEs on 2 nodes with 1 GPU each, to 1 PE

one PE. The rescaling functionality is exposed to the user in CharmStencil through a `rescale` call.

V. RESULTS

For our experiments we used the TACC Vista supercomputer. Each node on Vista has a single NVIDIA GH200 GPU with an Infiniband interconnect. We used a `ucx-linux-arm8-cuda` non-SMP Charm++ build.

A. Pipelined execution performance

CharmStencil executes the frontend and backend in a pipelined fashion to hide the overhead of the Python frontend. To study how effectively this model hides the overhead, we evaluated the performance of the Laplace solver from Figure 2 with a varying DAG depth. The experiment used a grid size of $16k \times 16k$ per GPU across 4 GPUs on 4 nodes and ran for 10,000 iterations.

Figure 6 shows the runtime with varying DAG depths compared to the ideal runtime (i.e., without any Python overhead). For very small DAG depths, the runtime is high
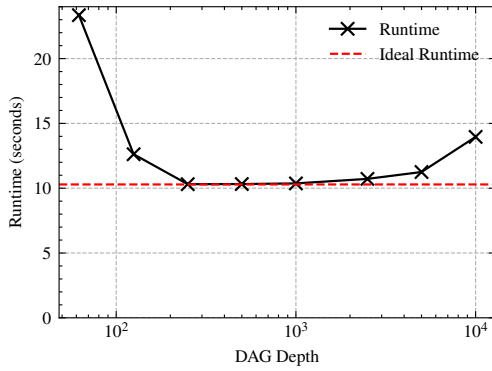
Fig. 6: Performance of a Laplace solver with varying DAG depth

due to frequent communication between the frontend and the backend. Conversely, a large DAG depth reduces this communication overhead but also reduces the overlap between frontend and backend execution. In Figure 6, a DAG depth of 10,000 results in the frontend executing the entire program before any operations are sent to the backend. Consequently, the total runtime includes the full Python overhead from the frontend, which was 3.7s in this example. Our results show that for a large range of DAG depths between these two extremes, our execution model effectively hides the Python overhead.

B. Scaling performance

We measured weak scaling performance on two applications - a Laplace equation solver from figure 2, and a more complex CFD benchmark for simulating a cavity flow using the Navier-Stokes equations [2]. We compared our results with those of cuPyNumeric and Devito. For the Laplace equation solver, we also compared our results with a handwritten Charm++ and CUDA implementation. cuPyNumeric is a drop-in NumPy replacement library based on the Legion runtime system [3] that supports execution on multi-node, multi-GPU systems using UCX and GASNet. For our experiments, we used a version of cuPyNumeric built with UCX. Devito is a high-level, Python-based DSL for stencil computations [17]. The open-source version of Devito supports GPU execution using OpenACC and can run on systems with multiple GPUs on a single node, but it does not support multi-node execution. We measured the runtime for Devito with the `advanced` optimization mode.

For the Laplace equation solver, we used a $16k \times 16k$ grid per GPU with 1000 iterations. For the cavity flow solver, we used a grid size of $16k \times 16k$ per GPU and 50 iterations. Figure 7 shows the weak scaling results for both benchmarks. Since Devito only supports single-node execution and the Vista supercomputer has one GPU per node, Devito's results are limited to a single GPU.

The results show that CharmStencil is faster than cuPyNumeric by a factor of 6 for the Laplace solver and a factor of 10 for the cavity flow solver, exhibiting near-perfect weak scaling. On a single GPU, CharmStencil performs 1.7x better than

Devito for the Laplace solver and 3.2x better for the cavity flow benchmark. We also see that CharmStencil performs on par with a handwritten Charm++ and CUDA implementation of the Laplace equation solver while reducing the code size from 830 to just 30 LoC.

C. Rescaling overhead

Rescaling in Charm++ has four sources of overhead:

1) Load balancing: When shrinking, chares and their corresponding GPU-resident data are moved away from nodes that will be removed. When expanding, a load balancing step migrates chares to newly added nodes to rebalance the workload.
2) Checkpoint: The chare's CPU data is checkpointed to Linux shared memory, while the GPU-resident data is copied to the corresponding daemon process before the application restarts.
3) Restart: The application restart itself incurs a startup overhead.
4) Restore: The chare's CPU data is read from Linux shared memory, and the GPU-resident data is retrieved and copied from the daemon process back to the Charm++ PE after the restart.

To study the contribution of each source to the total overhead, we measured each stage of rescaling by shrinking and expanding a CharmStencil application within a SLURM allocation. We first studied the overhead on a varying number of nodes with a constant problem size per GPU. Figures 8a and 8b show the overheads from shrinking to half the number of nodes and expanding to double the number of nodes, respectively. For the shrink tests, we used a data size of 1GB per GPU; for the expand tests, we used 2GB per GPU. In both cases, the results show that the overhead was dominated by the application restart time, with the total overhead on the order of one second.

To study the effect of data size on the rescaling overhead, we measured the cost of shrinking from 16 to 8 nodes with data sizes varying from 250MB to 4GB per GPU. We observed that the checkpoint, restore, and load balancing overheads scale linearly with the data size; however, the total overhead remains on the order of one second.

D. Performance with rescaling

To demonstrate the resource elasticity of CharmStencil, we used the Laplace solver with a $32k \times 32k$ grid distributed across 8 nodes. The application ran for 1000 iterations, after which we called `rescale` to scale the job down to 4 nodes. After another 1000 iterations, a second `rescale` call expanded the job back to 8 nodes. The runtime was measured every 10 iterations on the backend, as shown in Figure 9. As expected, the time per iteration doubles after the first rescale event because the application is running on half the number of GPUs. Following the second `rescale` call, the time per iteration returns to its original value as the application expands back to its initial 8-GPU allocation.
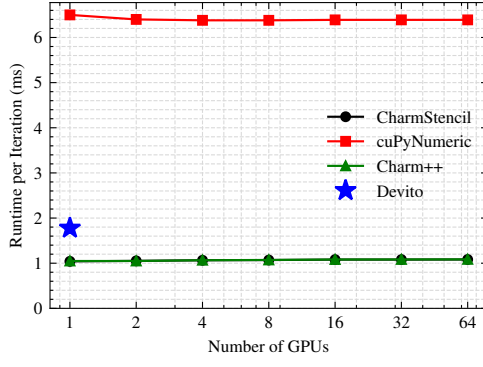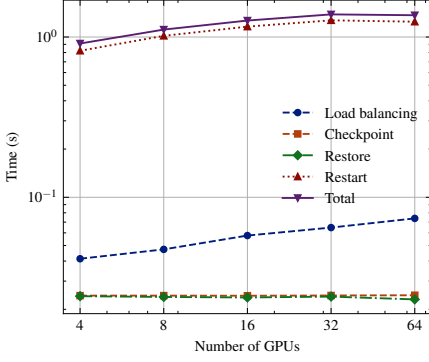
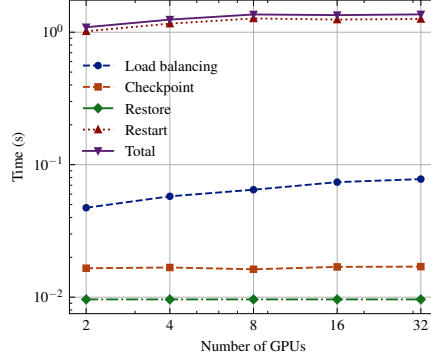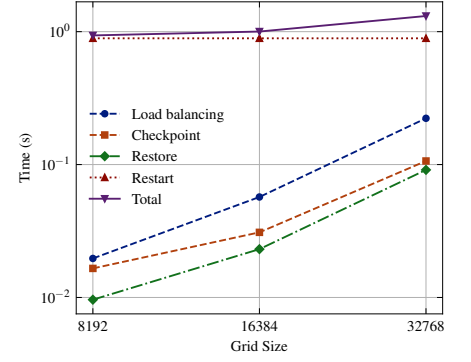(a) Laplace equation
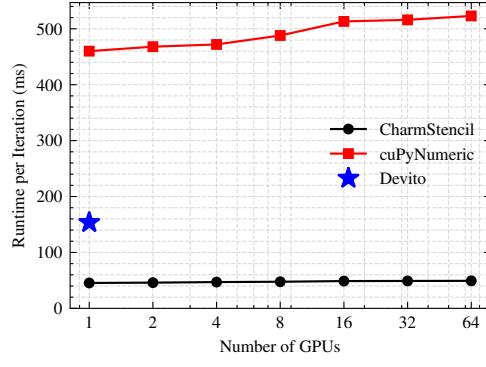
(b) Cavity flow

Fig. 7: Weak scaling results

(a) Shrinking to half the number of nodes. The x-axis shows the number of nodes before shrinking

(b) Expanding to double the number of nodes. The x-axis shows the number of nodes before expanding

(c) Shrinking from 16 to 8 nodes for different problem sizes. Grid size is the size of one dimension of the 2D grid per GPU

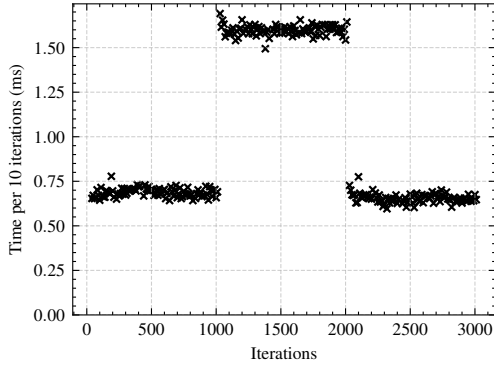Fig. 8: Contribution of different stages of rescaling to the total rescaling overhead

Fig. 9: Time per 10 iterations of a Laplace solver with 2 rescaling calls, from 8 to 4 nodes, and back from 4 to 8 nodes

VI. RELATED WORK

Several DSLs have been developed for stencil computations, broadly classified into two categories: high-level stencil abstractions and general NumPy drop-in replacements.

A. NumPy drop-in replacement DSLs

CuPy is a direct NumPy replacement for single GPUs, leveraging NVIDIA's highly optimized libraries (cuBLAS, cuDNN, etc.) for array operations [21]. cuPyNumeric is a distributed replacement that internally uses CuPy for array operations and the Legion runtime for managing distributed execution [3]. cuPyNumeric shows excellent weak scaling performance on various benchmarks, including CFD applications, using optimizations like control replication to minimize scheduling overhead and kernel fusion to improve data reuse.

B. Specialized stencil DSLs

Devito is a high-level, Python-based symbolic DSL for stencil computations [17]. Rather than the NumPy API, Devito uses a higher-level mathematical representation of equations, which it internally discretizes and JIT-compiles. Devito targets shared-memory (OpenMP), distributed-memory (MPI), and GPU (OpenACC) systems.

OPS is a C/C++ and Fortran-based DSL for computations on structured meshes [22]. Users write computations using data blocks and parallel loop constructs; OPS then uses a source-to-source translator to generate high-performance parallel code

for shared-memory CPUs, distributed-memory CPUs, and GPUs.

VII. CONCLUSION

In this paper, we present CharmStencil, a distributed, adaptive GPU stencil abstraction for Charm++. Existing stencil DSLs often force a trade-off between NumPy-like syntax—requiring significant porting effort—and performance. CharmStencil addresses this gap by providing a familiar NumPy-like frontend with a highly optimized backend.

A key challenge in Python HPC is interpreter overhead. We show that CharmStencil effectively hides this overhead using a pipelined execution model. Furthermore, our performance evaluations demonstrate that CharmStencil significantly outperforms both a generalized NumPy replacement and a specialized stencil abstraction, and performs on par with a handwritten Charm++ CUDA implementation.

Resource elasticity is increasingly becoming a necessity on modern supercomputing systems and cloud platforms. We extended the rescaling support in Charm++ to GPUs, incorporating elasticity into the CharmTyles framework. CharmStencil, built on CharmTyles, efficiently supports this feature. We demonstrated this by dynamically scaling an application, measuring the overhead of rescaling at approximately one second on a SLURM-based supercomputer.

VIII. ACKNOWLEDGMENTS

REFERENCES

[1] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. SC, 2014.

[2] Lorena Barba and Gilbert Forsyth. Cfd python: the 12 steps to navier-stokes equations. *Journal of Open Source Education*, 2(16):21, 2019.

[3] Michael Bauer and Michael Garland. Legate NumPy: Accelerated and Distributed Array Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Aditya Bhosale, Kavitha Chandrasekar, Laxmikant Kale, and Sara Kokkila-Schumacher. An elastic job scheduler for hpc applications on the cloud. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25)*, page 12, New York, NY, USA, November 2025. ACM.

[5] Aditya Bhosale, Zane Fink, and Laxmikant Kale. An abstraction for distributed stencil computations using charm++. In *Asynchronous Many-Task Systems and Applications: Second International Workshop, WAMTA 2024, Knoxville, TN, USA, February 14–16, 2024, Proceedings*, page 123–134, Berlin, Heidelberg, 2024. Springer-Verlag.

[6] Aditya Bhosale, Laxmikant Kale, and Sara Kokkila-Schumacher. Efficient and Cost-Effective HPC on the Cloud. In *The 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25)*, New York, NY, USA, jul 2025. ACM.

[7] Mohak Chadha, Jophin John, and Michael Gerndt. Extending slurm for dynamic resource-aware adaptive batch scheduling. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 223–232, 2020.

[8] J. Choi, D. F. Richards, and L. V. Kale. Achieving computation-communication overlap with overdecomposition on gpu systems. In *2020 IEEE/ACM 5th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pages 1–10, 2020.

[9] Jaemin Choi, Zane Fink, Sam White, Nitin Bhat, David F. Richards, and Laxmikant V. Kale. Gpu-aware communication with ucx in parallel programming models: Charm++, mpi, and python. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 479–488, 2021.

[10] Filippo Gioachin, Chee Wai Lee, and Laxmikant V. Kalé. Scalable Interaction with Parallel Applications. In *Proceedings of TeraGrid'09*, Arlington, VA, USA, June 2009.

[11] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. The MIT Press, 11 1999.

[12] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. Elasticflow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 266–280, New York, NY, USA, 2023. Association for Computing Machinery.

[13] Abhishek Gupta, Bilge Acun, Osman Sarood, and Laxmikant V. Kale. Towards Realizing the Potential of Malleable Parallel Jobs. In *Proceedings of the IEEE International Conference on High Performance Computing*, HiPC '14, Goa, India, December 2014.

[14] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with numpy. *Nature*, 585(7825):357–362, Sep 2020.

[15] Dominik Huber, Maximilian Streubel, Isaías Comprés, Martin Schulz, Martin Schreiber, and Howard Pritchard. Towards dynamic resource management with mpi sessions and pmix. In *Proceedings of the 29th European MPI Users' Group Meeting*, EuroMPI/USA '22, page 57–67, New York, NY, USA, 2022. Association for Computing Machinery.

[16] Jan Hönig and Martin Bauer. pystencils - Automatic Generation, Optimization and Analysis of Stencil Codes. In *ISC High Performance 2018*, 2018.

[17] Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul H. J. Kelly, Felix J. Herrmann, and Gerard J. Gorman. Architecture and performance of devito, a system for automated stencil computation. *ACM Trans. Math. Softw.*, 46(1), apr 2020.

[18] Daniel Medeiros, Jacob Wahlgren, Gabin Schieffer, and Ivy Peng. Kub: Enabling elastic hpc workloads on containerized environments. In *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 219–229, 2023.

[19] Daniel J. Milroy, Claudia Misale, Giorgis Georgakoudis, Tonia Elengikal, Abhik Sarkar, Maurizio Drocco, Tapasya Patki, Jae-Seung Yeom, Carlos Eduardo Arango Gutierrez, Dong H. Ahn, and Yoonho Park. One step closer to converged computing: Achieving scalability with cloud-native hpc. In *2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pages 57–70, 2022.

[20] Ian Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[21] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[22] István Z. Reguly, Gihan R. Mudalige, and Michael B. Giles. Loop tiling in large-scale stencil codes at run-time with ops. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):873–886, 2018.

[23] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Zachary J. Weiner. Stencil Solvers for PDEs on GPUs: An Example From Cosmology. *Computing in Science & Engineering*, 23(4):55–64, 2021.