Autonomous Execution for Multi-GPU Systems: CPU-Free Blueprint and Compiler Support

by

Javid Baydamirli

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

KOÇ ÜNİVERSİTESİ

September 22, 2023

Autonomous Execution for Multi-GPU Systems: CPU-Free Blueprint and Compiler Support

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Javid Baydamirli

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Assoc. Prof. Dr. Didem Unat (Advisor)

_____

Assoc. Prof. Dr. Alptekin Küpçü

_____

Prof. Dr. Can Özturan

Date: _____

ABSTRACT

Autonomous Execution for Multi-GPU Systems: CPU-Free Blueprint and Compiler Support
Javid Baydamirli
Master of Science in Computer Science and Engineering
September 22, 2023

As multi-GPU systems become more prolific in the field of supercomputing, scientific applications are adapted and scaled up to take advantage of the highly parallel accelerators for increased performance. However, the traditional model of GPU programming leaves much to be desired in multi-GPU settings, wherein communication among devices - one of the largest points of contention and bottlenecks in scientific applications - is controlled by the CPU. This kind of one-sided control leads to undue latencies incurred by the constant back-and-forth of synchronization and API calls between the host and devices, and harms application scaling as the number of GPUs grows.

This work first proposes the fully autonomous *CPU-Free* execution model for multi-GPU applications that completely excludes the involvement of the CPU beyond the initial kernel launch. We systematically combine several techniques such as persistent kernels, thread block specialization, and GPU-initiated communication and synchronization to significantly reduce host-incurred latencies and facilitate further optimizations. We benchmark our proposed model on a broadly used iterative solver, 2D/3D Jacobi Stencil and improve 3D stencil communication latency by 58.8% compared to CPU-controlled baselines on 8 NVIDIA A100 GPUs.

The second part of this work adds compiler support to easily write performant *CPU-Free* code in high-level Python by extending the DaCe framework with GPU-centric communication intrinsics. We compare automatically generated CPU-Free code to existing distributed facilities in DaCe and observe over 96% performance improvement in Stencil benchmarks.

ÖZETÇE

Çoklu GPU Sistemleri İçin Otonom Yürütme: CPU'suz Tasarım ve Derleyici Desteği

Javid Baydamirli

Bilgisayar Bilimleri ve Mühendisliği, Yüksek Lisans

22 Eylül 2023

Çoklu GPU sistemlerinin süper bilgisayar alanında daha yaygın hale gelmesiyle birlikte, bilimsel uygulamalar, artan performans için yüksek paralel hızlandırıcılardan faydalanmak amacıyla uyarlanmakta ve ölçeklendirilmektedir. Ancak, GPU programlamasının geleneksel modeli, bilimsel uygulamalardaki en büyük sorun ve darboğazlardan biri olan cihazlar arası iletişimi, çoklu GPU ortamlarında bırakılmak istenilen birçok şeyi geride bırakmaktadır - bu tür tek taraflı kontrol, sürekli olarak ana işlemci (CPU) ile cihazlar arasında senkronizasyon ve API çağrıları arasındaki gidip gelmenin neden olduğu aşırı gecikmelere yol açar ve cihaz sayısı arttıkça uygulama ölçeklenmesine zarar verir.

Bu çalışma ilk olarak, çoklu GPU uygulamaları için, ilk başlatmanın ötesinde CPU'nun katılımını tamamen dışlayan, tamamen özerk CPU'suz yürütme modelini önermektedir. Bu amaçla *Persistent kernel*'ler, Thread Block özelleştirme, GPU-başlatılan iletişim ve senkronizasyon gibi çeşitli teknikleri sistematik olarak birleştirmekle ana işlemci kaynaklı gecikmeleri önemli ölçüde azaltıyor ve diger optimizasyonları etkinleştiriyoruz. Önerdiğimiz modeli, yaygın olarak kullanılan 2D/3D Jacobi Stencil iteratif çözücüde test ediyoruz ve 8 NVIDIA A100 GPU üzerinde CPU tarafından kontrol edilen temellere kıyasla 3D Stencil iletişim gecikmesini $\%58,8$ oranında iyileştiriyoruz.

Bu çalışmanın ikinci bölümü, Python'da performanslı *CPU'suz* kod yazmayı kolaylaştırmak için derleyici desteği ekler ve DaCe çerçevesini GPU-merkezli iletişim özellikleri ile genişletir. Otomatik olarak oluşturulan CPU'suz kodu, DaCe'deki mevcut dağıtık kodlarla karşılaştırır ve Stencil testlerinde %96'dan fazla performans iyileştirmesi görürüz.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

2D/3D	2/3-Dimensional
API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DSL	Domain Specific Language
GPU	Graphics Processing Unit
HPC	High Performance Computing
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
P2P	Peer to Peer
PE	Processing Element
SM	Streaming Multiprocessor
TB	Thread Block

Chapter 1

INTRODUCTION

The significance of accelerated computing has increased in recent years, as many modern HPC systems now equip multi-GPU nodes [Meuer et al., 2023]. Scaling up applications to exploit such heterogeneity is difficult, however, as communication among devices can quickly become a performance bottleneck [Shimokawabe et al., 2011], resulting in poor scaling. The traditional model of GPU programming where the host serves as the orchestrator of execution by launching kernels, issuing communication, and managing the synchronization of devices, we argue, is a significant cause of many undue latencies that can be eliminated by granting this control to the devices.

Though data transfers among devices are more efficient now thanks to advancements in both programming models and GPU hardware, such as peer-to-peer communication [NVIDIA, 2011a], direct GPU interconnents [Foley and Danskin, 2017], and GPUDirect RDMA [Hamidouche et al., 2015] that create a direct path for data from one device to another without the need for CPU-side buffers, these transfers are still *initiated* by the CPU through host-managed streams and runtime API calls. On top of controlling communication, the host is additionally responsible for both in-device and peer synchronization, routing data transfers, and overlapping computation and communication through the aforementioned means that require kernels to halt their execution and synchronize with their host in each step.

This work first addresses the issues outlined above and introduces a new programming model for communication-bound multi-GPU applications, the *CPU-Free Model*, that gives devices full autonomy over their entire execution. We leverage several techniques such as persistent kernels [Gupta et al., 2012], specialized/co-

operative thread blocks, GPU-side global synchronization, and GPU-initiated data transfers through P2P load/stores and NVSHMEM - a GPU-initiated communication library [NVIDIA, 2022b] - that allow us to eliminate host-incurred latencies and better utilize device resources. We implement 2D and 3D Jacobi Stencils [Spotz and Carey, 1995] in the *CPU-Free Model* as our proof of concept and demonstrate significantly improved communication overheads and overall execution times compared to *CPU-Controlled* baselines.

Later, considering that writing multi-GPU code in an emerging programming model can be a daunting task, we generalize the proposed model in a popular high-level parallel programming framework, *DaCe*, [Ben-Nun et al., 2019]. We extend DaCe to support the CPU-Free Model with new GPU-centric communication library nodes and trivially port distributed DaCe benchmarks [Ziogas et al., 2021] to the CPU-Free model. We again observe significant performance improvements compared to the baseline in single node environments.

Overall, our contributions are as follows:

- We introduce the *CPU-Free Model* - a multi-GPU execution model that that reduces latencies and achieves better communication overlap by eliminating host involvement - and detail its core components and implementation.

- We describe and implement iterative 2D and 3D Jacobi Stencils in the proposed model and provide a blueprint for writing CPU-Free code.

- We evaluate the performance of the implemented applications against CPU-controlled baselines with various levels of device autonomy.

- We extend DaCe with CPU-Free support to facilitate writing high-level Python code that compiles to performant CPU-Free device code.

Chapter 2

BACKGROUND & MOTIVATION

In this chapter, we present background information on multi-GPU communication and the traditional model of execution in the context of iterative stencil solvers. We then quantify and detail the shortcomings of the said model and motivate this research. Additionally, we discuss the DaCe framework, its internals and code generation.

2.1 Stencil Computation

Iterative Stencil Loops are a widely used technique with many applications in HPC such as fluid simulations and solving partial differential equations. As an iterative method, the computation involves regularly updating a structured grid [Meneghin et al., 2022] in each time step . The Jacobi method in particular - the subject of the experiments in this work - solves the 2D-Laplace equation, formulated as follows [NVIDIA, 2022a].

$$\Delta u(x, y) = 0 \forall (x, y) \in \Omega \ \delta\Omega$$

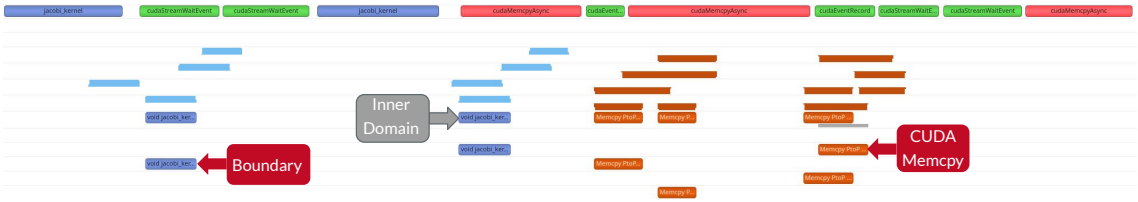
2.2 Multi-GPU Stencil

We present a high-level overview of a typical implementation of a CPU-controlled multi-GPU stencil in Listing 2.1a as provided by NVIDIA [NVIDIA, 2022a].

In the absence of GPU-side global barriers, the CPU first ❶ maintains a time loop that repeatedly launches the kernels in every iteration. Communication and computation overlap is achieved by splitting the program into two kernels that are enqueued in concurrent CUDA streams - `comp_stream`, and `comm_stream`. As such, the CPU ❷ launches the compute kernel performing the stencil operations in a `comp_stream`, and asynchronously performs ❸ synchronization with neighbors. Next, the CPU

```
// ❶ Time loop on CPU
while (iter++ < num_iterations) {
    // ❷ Launch compute kernel
    stencil_kernel<<<..., comp_stream>>>(...)
    // ❸ Sync with neighboring GPUs
    wait_neighbors(comm_stream)
    // ❹ Compute and communicate boundaries
    compute_boundaries<<<..., comm_stream>>>(...)
    write_neighbors(comm_stream)
    // ❺ Sync comm and comp streams
    sync(comm_stream, comp_stream)
}
```

(a) CPU-Controlled Overlapping Stencil Overview

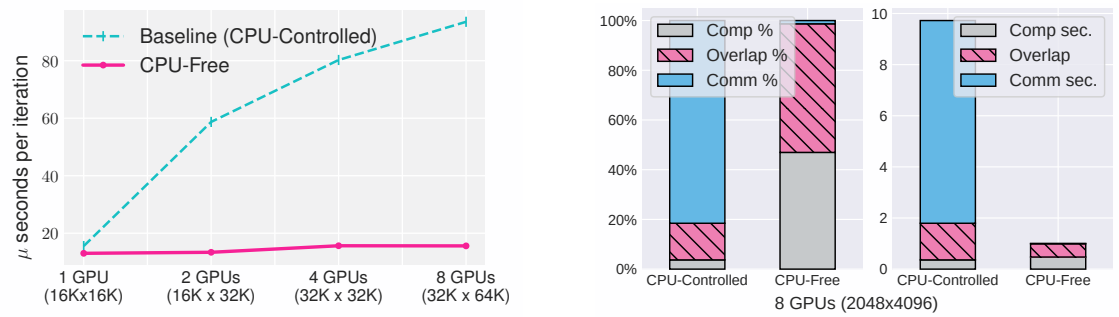


(b) NVIDIA Nsight Timeline at 8 GPUs

Figure 2.1: CPU-Controlled Overlapping Stencil Overview and Nsight Timeline

❹ launches a kernel to compute the boundary rows and communicate them to neighbors as halos, again in the `comm_stream`, to achieve overlap. Thus, step ❷ is overlapped with steps ❸ and ❹. Finally, the CPU waits on these two streams before advancing to the next iteration and synchronizes with neighboring peers. It should be noted that in the real implementation, this final step is achieved with CPU-side synchronization methods such as OpenMP and MPI barriers [NVIDIA, 2022a].

Figure 2.2 shows the ratio of computation to communication as well as the overlapped portion of the stencil code detailed above in a small domain. We observe that although the program takes advantage of CUDA streams for hiding communication latency, the synchronization and kernel latencies, and the CPU-initiated data trans-



(a) Pure communication overhead

(b) Communication overlap ratio

Figure 2.2: (a) Communication and synchronization overheads with no computation (b) Communication overlap ratio % and total execution time in seconds

fer overhead - even in the presence of a direct data path among devices - are too high to fully overlap, resulting in much worse scaling. As a result, communication takes 96% of the execution time, of which only 19% is overlapped with computation. The CPU-Free version, in contrast, has much lower total overheads, and manages to hide almost the entirety of the communication latency. Though this kind of high communication latency can be hidden more easily in large domains when computation takes a bigger portion, CPU-free execution is of significant interest in strong scaling cases and in simulation phases where the workload drops, such as boundary conditions for 2D planes in a 3D discretized domain to solve PDEs [Afanasyev et al., 2021, Yamaguchi et al., 2017, Wahib and Maruyama, 2014].

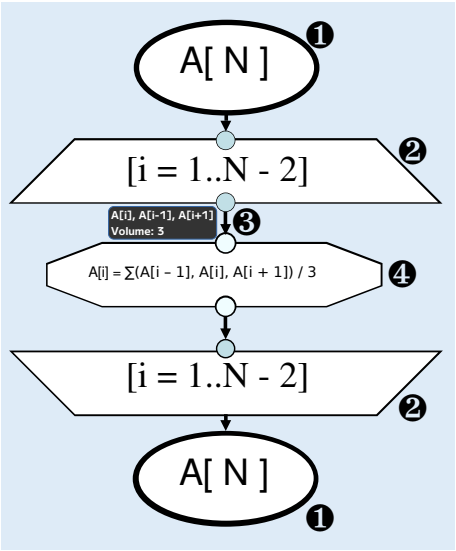
2.3 Data Centric Parallel Programming

DaCe is an intermediate representation and a multi-target compiler with a Python frontend that focuses on data-centric transformations and performance portability [Ben-Nun et al., 2019, SPCL, 2023]. At the core of DaCe is the Stateful DataFlow multiGraph IR constructed from high-level Python code with several extensible components used in this work to generate CPU-Free code.

A basic DaCe program is an annotated Python function 2.3a, compiled to an SDFG intermediate representation 2.3b. The core components of the SDFG are as

```python
import dace as dc

@dc.program
def func(A[N]):
    A[1:−1] = (A[:−2] + x[1:−1] + x[2:]) / 3

if __name__ == '__main__':
    N = 1024
    A = np.random.rand(N)

    sdfg = func.to_sdfg()
    sdfg = auto_optimize(sdfg, GPU)

    sdfg(A=A)
```

(a) DaCe program written in Python



(b) Generated SDFG IR (simplified)

Figure 2.3: Sample DaCe program

follows

- ❶ **Access nodes** point to arrays/containers used in the DaCe program. Outgoing edges indicate read accesses, while incoming edges represent writes.
- ❷ **Maps** represent data parallelism in a DaCe program. Map nodes have symbolic ranges as shown in 2.3b and can be scheduled to CPU threads or parallel hardware, like GPUs.
- ❸ **Memlets** indicate data movement between nodes and include information on the data at hand such as the volume, subset and write conflict resolution.
- ❹ **Tasklets** represent arbitrary computation within a given memory connection.
- **Library nodes** are high-level constructs that represent specific functions such as BLAS operations, MPI calls, and so on. Library nodes expand to the

components listed above and can include several specializations. We primarily implement library nodes to generate CPU-Free code.

Once built, `Transformations` can be applied on the SDFG, both to its entirety and to subgraphs, using pattern matching. We utilize this facility to resolve certain prerequisites to CPU-Free execution, as will be discussed in later chapters.

Chapter 3

CPU-FREE EXECUTION MODEL

This chapter introduces the foundation of this work - the *CPU-Free Execution Model*, its core components, benefits over traditional execution, and a Stencil use-case as a proof of concept. The execution model we propose makes use of several prerequisites and techniques, such as persistent kernels, thread-block specialization, GPU-initiated data transfers, and device-side synchronization to completely remove the CPU from the control path, as discussed below.

3.1 Overview

3.1.1 Persistent Kernels

GPU kernels have traditionally been implemented in a bulk-synchronous manner - also referred to as *discrete kernels* in this work. In the case of iterative solvers, discrete kernels are also 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. Moreover, synchronization between timesteps required in many iterative applications to resolve temporal dependencies is implemented on the host side with implicit synchronization [NVIDIA et al., 2020].

In order to provide GPU more autonomy in such applications, we make use of persistent kernels [Gupta et al., 2012] 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 [Gupta et al., 2012, Zhang et al., 2022, Zhang et al., 2020], there are wider implications of persistent execution, especially in multi-GPU scenarios where communication has traditionally been initiated outside

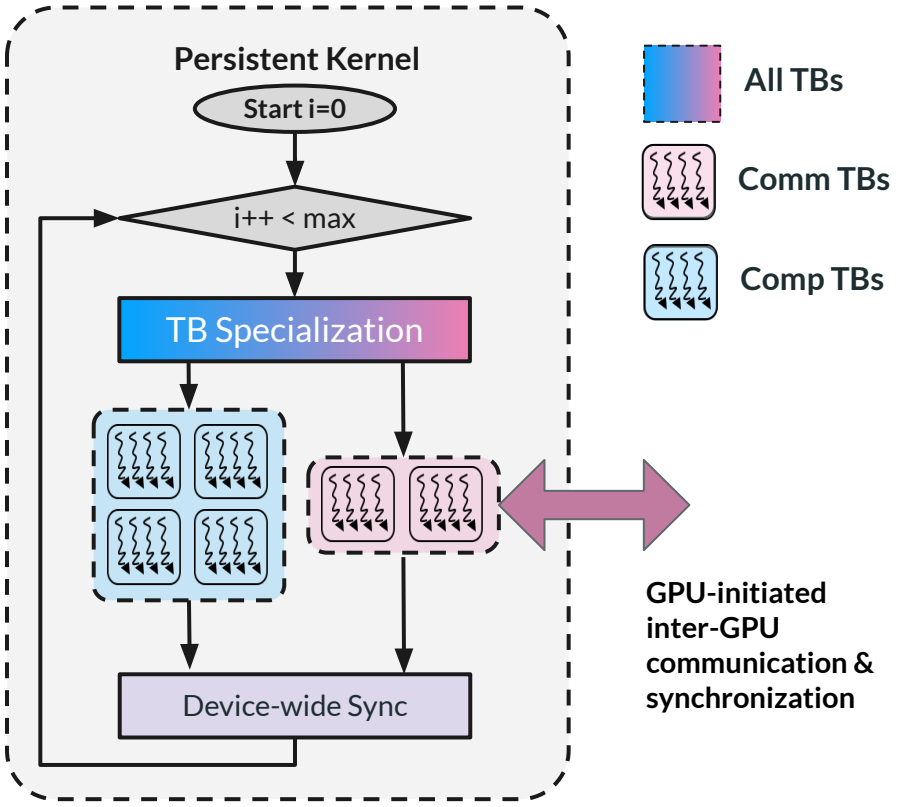


Figure 3.1: CPU-free execution model leverages persistent kernels, thread-block (TB) specialization, GPU-initiated data transfers, and device-side synchronization. Comm: Communication, Comp: Computation.

of devices, as discussed below.

3.1.2 Device-Side Synchronization

Moving the time loop into the device necessitates the need for in-kernel global synchronization for the temporal dependencies mentioned previously. Such in-kernel synchronization had been limited to a single thread block at most prior to CUDA 9.0 [NVIDIA, 2011b], 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 [Zhang et al., 2020], however, maintaining a single kernel throughout the computation is desirable, as more caching

optimizations and better shared memory utilization, 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. We again delegate this task to the GPUs using device-side barriers and point-to-point synchronization with NVSHMEM [NVIDIA, 2022b].

3.1.3 Thread Block Specialization

In order to implement concurrency within a persistent kernel in the absence of streams, we treat thread blocks as individual execution units and specialize them for asynchronous tasks, in contrast to their traditional data-parallel usage. Similar to warp specialization described in [Bauer et al., 2014a], we assign concurrent sub-tasks to specific colocated thread blocks and synchronize them with the aforementioned Cooperative Groups facilities. In particular, for Stencil computation, we use TB Specialization to overlap communication and computation where a number of blocks in a kernel manage communication and disjoint boundary region computation, while the remainder of the kernel handles the inner region of the grid.

3.1.4 GPU-Initiated Data Movement

We allow devices to not only move their data independently among peers but also *initiate* the transfers autonomously within a long-running persistent kernel, in lieu of host-initiated memory operations. Such CPU-Free data movement can be implemented with direct loads and stores to and from peer devices' memories using UVA [NVIDIA et al., 2020], or with the GPU-centric implementation of OpenSHMEM [Poole et al., 2011], NVSHMEM, provided by NVIDIA [Potluri et al., 2017, NVIDIA, 2022b]. This work mainly explores the latter, as it provides an optimized API for fine-grained GPU-initiated communication as well as Cooperative Groups support.

3.2 Benefits of CPU-free Execution

We observe the following benefits of the proposed CPU-Free execution model in multi-GPU applications compared to traditional CPU-controlled execution.

1. **Reduced API overheads** Launching a single kernel that encapsulates all computation and communication operations eliminates overheads incurred by host-side API calls to CUDA runtime and possible synchronization required by them.

2. **Reduced communication latency** As an extension of the previous point, GPU-initiated communication has significantly smaller latency compared to CPU-controlled baselines (Figure 2.2). Being able to communicate and synchronize with peer GPUs without the need to synchronize with the host allows kernels to begin transfers as soon as the data is ready, enabling more asynchrony.

3. **Better overlap and communication hiding** Moreover, thanks to reduced overall communication time, it is easier to overlap communication with computation, which is most notable in strong scaling scenarios where execution time becomes increasingly dominated by communication latency. We observe that the CPU-Free model achieves a great degree of overlap when CPU-controlled baselines struggle (Figure 2.2).

4. **Shared memory usage across iterations** As a traditional CPU-controlled iterative program launches kernels for every time step, the GPU shared memory, whose lifetime is that of the kernel, cannot be utilized to cache intermediate results between iterations, necessitating reads and writes from slower global memory. For memory-bound applications, reducing such accesses has a significant performance benefit, as shown by [Zhang et al., 2022], who utilize the large number of registers and shared memory in a single-GPU persistent kernel to cache a portion of the domain. We implement the techniques discussed

in their work and extend it to multi-GPU for the stencil methods detailed in later chapters (Figure 6.1).

Chapter 4

CPU-FREE STENCIL

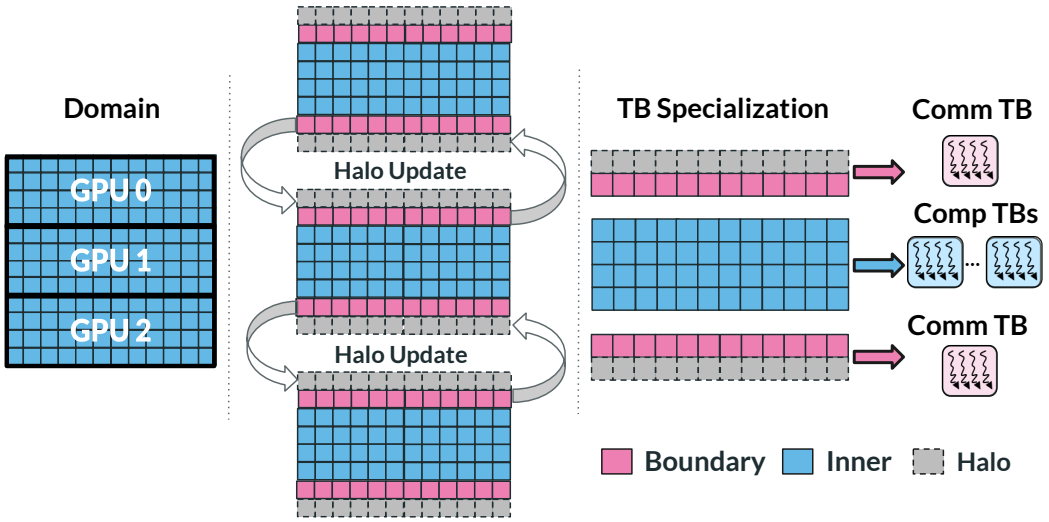


Figure 4.1: Domain decomposition, halo updates, and thread block (TB) specialization for 2D5pt stencil. While communication (comm) TBs handle the boundary computation and halo update with neighbors, computation (comp) TBs handle the inner domain.

Iterative stencil solvers [Strikwerda, 2004] are natural candidates for the proposed execution model as their temporal dependence across the domain requires data movement among all devices and synchronization at each iteration. Thus, we present a CPU-free implementation of multi-GPU Jacobi stencils with overlapped communication as our use case.

The stencil domain can be divided equally among GPUs and split into two independent parts in each device: the inner domain and the boundary, as illustrated in Figure 4.1. Halo regions that refer to values in neighboring GPUs' domains are

```
__global__ void CPU_Free_Jacobi(...) {
    // Time loop on GPU
    while (iter++ < num_iterations) {
        //ⓐ Compute boundary using top neighbor's values
        if (TB_index == 0) {
            //❶ Wait for top neighbor to signal
            wait_top_neighbor(...)
            //❷ Compute top boundary using halos
            top_boundary = compute(top_halo, south, ...)
            //❸ Write to top neighbor's bottom halo
            write_top_neighbor(top_boundary)
            //❹ Signal top neighbor that iteration is done
            signal_top_neighbor(...)
        }
        //ⓑ Compute boundary with bottom neighbor's values
        if (TB_index == 1) { ... }
        //ⓒ Remaining TBs compute the inner domain
        if (TB_index == <rest>) { compute_inner() }
        //❺ Synchronize all TBs in kernel
        grid.sync()
}}
```

Listing 4.1: Stencil kernel using CPU-Free model

appended to each chunk and require communication at each time step. As proposed in [Shimokawabe et al., 2011], we can compute the boundary region independently of the inner domain and communicate concurrently with neighbors while the bulk of the inner domain is being processed.

The overlapping design we propose utilizes a persistent kernel that specializes a number of thread blocks to compute the boundary region and issue communication routines as shown in Listing 4.1. We move the time loop inside the kernel and specialize two concurrent thread blocks for the boundary region (ⓐ, ⓑ) and the remainder of the device (ⓒ) for the inner domain. Each of the boundary thread blocks ❶ initially waits on a flag for their corresponding neighbor to signal the availability

of the halo region, and ❷ uses the said values to compute its boundary region. They then ❸ proceed to commit the new values into buffers in their neighbors' memories, and ❹ signal them of their availability. ❺ Finally, all thread blocks are synchronized at the end of the loop before continuing to the next iteration.

An alternative design to specializing thread blocks in one kernel is to have two co-resident persistent kernels in separate streams, managing boundary and inner domain processing independently. This kind of configuration is more modular and easier to adapt to existing single-GPU kernels but requires an extra sync point between the local pairs of streams in each GPU. We did not observe any significant performance improvement or degradation from this design compared to the single-stream version.

Iterative methods, as discussed earlier, benefit greatly from long-running kernels, as the intermediate results can be cached to be used in subsequent time steps instead of being committed to global memory every iteration. PERKS by Zhang et al. [Zhang et al., 2022] demonstrates this by explicitly caching a portion of the domain in registers and shared memory across iterations through a priority-based caching scheme. We apply our communication scheme on top of the single-GPU stencil implementation of PERKS and extend it with minimal intrusions to the upstream kernel.

4.1 Implementation

We based our stencil code on NVIDIA's open sources multi-gpu programming models repository [NVIDIA, 2022a]. Though the samples do not necessarily have the most optimized computational kernels - the repository focuses exclusively on communication methods - we refer to it as our main baseline for direct comparisons of communication overhead.

4.1.1 Synchronization and Halo Exchange

In order to synchronize all thread blocks at time steps, we launch our kernels using the CUDA cooperative groups API, which provides the device-side `grid.sync()`

call. We utilize signaling operations provided by the NVSHMEM API [NVIDIA, 2022b] for inter-GPU synchronization. Pairs of flags are allocated in the symmetric heap for top and bottom neighbors - four in total for each processing element - using `nvshmem_malloc()`. The flags are waited on using `nvshmem_signal_wait_until()` in boundary thread blocks before computation begins to ensure neighboring devices have committed values of the previous time step. The signaling flow is essentially a semaphore wherein neighboring devices signal the availability of halos of a given time step by setting the corresponding flag to the value of the next iteration, while waiting is done by comparing the flag to the current iteration. The signaling operation is implemented using the composite `nvshmemx_putmem_signal_nbi_block()` call that asynchronously performs data movement and subsequently updates a given flag at the destination on completion. This API is also used for writing to halo regions that reside in neighboring devices' memories. Synchronizing local concurrent kernels, if needed, is done by busy waiting on a flag in local device memory.

4.1.2 Thread Block Specialization

We specialize thread blocks to carry out inner computation, boundary computation, and communication. In order to balance the two phases of execution, we adjust the number of thread blocks to reserve for boundary computation with the domain size. We use the following formula to determine work allocation.

$$\texttt{boundary\_TB\_num} = \frac{\texttt{TB\_total} * \texttt{boundary\_size}}{\texttt{inner\_size} + 2 * \texttt{boundary\_size}}$$

$$\texttt{inner\_TB\_num} = \texttt{TB\_total} - 2 * \texttt{boundary\_TB\_num}$$

where `TB_total` is the total number of thread blocks available in the device for the given thread count.

Splitting the thread blocks proportionally to the amount of work is necessary for smaller and unbalanced 3D domains to achieve proper overlap, as they are susceptible to being bound by the boundary region computation and communication time otherwise.

4.1.3 PERKS Integration

We can apply our communication scheme to existing single-GPU kernels by swapping them into our inner-computation kernel. We extend a single-GPU stencil kernel provided by PERKS [Zhang et al., 2022], as discussed in Section 4.1, as it is a highly optimized persistent implementation with explicit caching. We partition the domain as shown in Figure 4.1 and restrict the PERKS kernel to the inner domain while keeping it oblivious to the multi-GPU nature of execution, treating it mostly as a black box. The PERKS kernel is minimally modified to synchronize with the communication stream at the end of each iteration, and the domain resides in memory buffers shared by both. Since both kernels access the same buffers in device memory, we need to make PERKS process a disjoint portion without memory conflicts, which is done by making it treat the boundary layers as immutable halos. As PERKS excludes read-only halos in its caching strategy, the kernel freely observes memory writes from the boundary stream by reading from the global memory in every iteration.

4.1.4 Limitations

The CPU-Free model, as a result of its persistent execution, requires thread blocks in kernels to be co-resident, meaning it is only possible to launch kernels with as many blocks as the device can concurrently run at a given time. This is a limitation set by the Cooperative Groups API, and disallows oversubscription of thread blocks to fit the domain, instead delegating this work distribution to the programmer. In large domains, such manual tiling can introduce inefficiencies compared to discrete kernel scheduling in CUDA.

Chapter 5

CPU-FREE CODE GENERATION WITH DACE

Our extensions to DaCe in this work focus primarily on enabling GPU-initiated communication in distributed data-centric applications. As such, the main contribution of this work is the addition of the GPU-initiated NVSHMEM communication library.

5.1 Persistent GPU fusion

DaCe provides a `GPUPersistentKernel` transformation that fuses a given GPU subgraph into a single persistent GPU kernel. The transformation works well for simple programs and efficiently fuses Maps[1], but handles branches and state changes conservatively, e.g., scheduling them in a single thread followed by a grid-wide barrier when global memory is accessed. We relax the barrier generation slightly, limiting it to subgraph edges, but do not address further inefficiencies in this work, and direct our efforts towards correct persistent code generation instead.

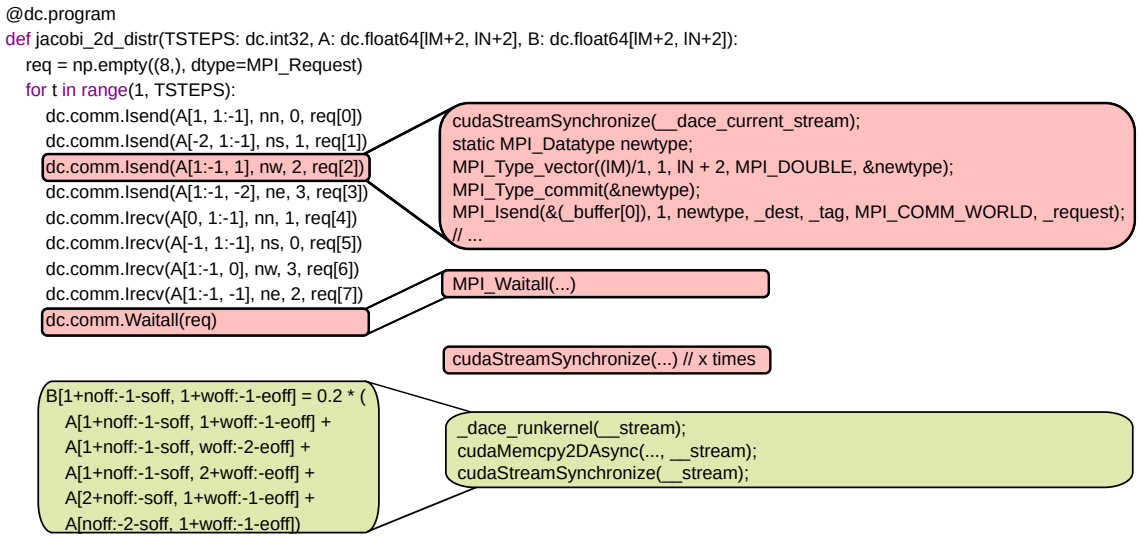
We additionally implement a specialization of DaCe's array-to-array copy routine - now generated inside device kernels - that utilizes GPU threads for better performance.

5.2 Existing distributed computing support

DaCe provides MPI library nodes with frontend support to allow programs to express communication directly in Python code [Ziogas et al., 2021]. As MPI communication is expressed in explicit nodes in the dataflow graph, they participate in the optimization process and can be modified within transformations and passes. Moreover, DaCe supports high-level multi-dimensional array operations in NumPy syntax

[1] Refers to Map nodes in DaCe. See 2.3

[Harris et al., 2020, Ben-Nun et al., 2019, Ziogas et al., 2021] in communication calls to further increase programmer productivity.



(a) MPI-based communication and generated code

(b) Nsight Timeline of communication calls and kernel launches

Figure 5.1: DaCe 2D Jacobi with strided access (MPI)

Listing 5.1a is an example of a distributed 2D Jacobi implementation utilizing DaCe's MPI operations [Ziogas et al., 2021]. We note the four pairs of MPI `Send` and `Recv` calls for north, south, east, and west neighbors of each rank. We also note the strided array accesses passed to MPI nodes expressed in high-level Python syntax.

The generated code highlighted in 5.1a includes several stream synchronize calls in every time step alongside the MPI calls as well as CPU-initiated `memcpy` operations. As a result, we observe little to no overlap in this example in the Nsight timeline 5.1b, as the program is dominated by host-side communication and synchronization calls.

5.3 NVSHMEM library and in-kernel expansion

We target a similar interface to the provided MPI library with the same level of abstraction in the NVSHMEM nodes. For the purposes of this work, we focus mainly on `nvshmem_putmem_*()` and `nvshmem_signal_*()` family of remote memory and signaling operations.

```
@dc.program
def jacobi_1d(A: dc.float64[N], ...):
    for t in range(1, TSTEPS):
        dc.comm.Isend(A[1], nw, 3, req[0])
        dc.comm.Isend(A[−2], ne, 2, req[1])
        dc.comm.Isend(A[0], nw, 2, req[2])
        dc.comm.Isend(A[−1], ne, 3, req[3])

        dc.comm.Waitall(req)

        A[1:−1] = ...
```

Listing (5.1) MPI-based communication

```
@dc.program
def jacobi_1d(A: dc.float64[N], ...):
    for t in range(1, TSTEPS):
        nvshmem.PutmemSignal(A[−1], A[1], flags[0], t, nw)
        nvshmem.PutmemSignal(A[0], A[−2], flags[1], t, ne)
        nvshmem.PutmemSignal(flags[0], t)
        nvshmem.PutmemSignal(flags[1], t)

        # ..

        A[1:−1] = ...
```

Listing (5.2) NVSHMEM replacements

Listing 5.3: Current DaCe distributed workflow versus revamped communication calls

To facilitate GPU-initiated synchronization alongside data transfers, we implement the composite `putmem_signal_*()` and `signal_wait_*()` calls that use flag-based atomic signaling for point-to-point synchronization [NVIDIA, 2022b]. These calls are necessary to ensure correctness without collective synchronization, and supersede `MPI_Send()` and `MPI_Recv()` family of calls with slightly different semantics, as shown in 5.2. The specifics of the usage of these calls are discussed in Chapter 4.1.

5.3.1 Strided and Single-element Access

In order to provide a uniform interface to our library regardless of the shape of arrays, we implement a compile-time check on the shapes of views passed to library nodes. As NVHSMEM provides specialized strided and single-element remote memory operations, namely `nvshmem_<TYPENAME>_iput_*()`/`nvshmem_<TYPENAME>_iget_*()` and

`nvshmem_<TYPENAME>_p()`, respectively, we dynamically switch the implementation of communication nodes to their specialized counterparts when appropriate, as shown in 5.6. It should be noted that as the aforementioned operations do not have combined signaling variants [NVIDIA, 2022b], we additionally generate manual signal (`nvshmem_signal_op()`) and memory ordering (`nvshmem_quiet()`) operations following the remote memory calls when converting from signaled nodes.

```
@dc.program
def jacobi(A: dc.float64[lM + 2, lN + 2]):
    # ...
    PutmemSignal(A[1:−1, −1], A[1:−1, 1], flags[2], t, nw)
```

Listing 5.4: Python code

```
double* _dest = gpu_A + ((2 * lN) + 3); // etc
nvshmem_double_iput(_dst, _src, lN + 2, lN + 2, (lM)/1, nw);
nvshmem_quiet();
nvshmemx_signal_op(_s_addr, t, NVSHMEM_SIGNAL_SET, nw);
```

Listing 5.5: Generated tasklet (simplified)

Listing 5.6: Code generation for strided remote memory operations

5.3.2 Scheduling and Overlap

Due to the limitations of the existing Persistent scheduling of DaCe discussed in Section 5.1, we currently schedule all GPU-initiated communication calls and signaling operations in a single thread followed by a grid sync. While it guarantees correct execution, this kind of conservative scheduling is not ideal, as there are limited opportunities for intra-kernel overlap. In order to ameliorate this, we expand to nonblocking variants of NVSHMEM memory operations, such as `nvshmem_putmem_nbi()` by default in our library nodes. Furthermore, NVSHMEM extended API calls such as `nvshmemx_putmem_block()` that cooperatively use multiple threads in a given thread blocks or a warp to transfer data, while implemented, are not scheduled correctly at this time. We additionally provide Mapped specializations of the aforementioned functions that expand to single-element `nvshmem_<TYPENAME>_p()` called by several GPU threads, as well as Map-less single-element nodes that can be manually placed in Maps by users. Though both of these options can be scheduled correctly in persistent kernels and utilize multiple threads and blocks, we focus on and report single-thread scheduled nonblocking signaled operation as detailed above.

5.3.3 PGAS Symmetric Memory Allocation

NVSHMEM requires buffers used by the remote memory API, both for storage and signaling, to be allocated on the symmetric heap [NVIDIA, 2022b]. In order to selectively allocate buffers as such, we add a new GPU storage type, `GPU_NVSHMEM` and add support to CUDA code generation for multiple global GPU storage types. We also add an `NVSHMEMArray` transformation that automatically sets Access nodes accessed by NVSHMEM library nodes to `GPU_NVSHMEM`.

5.4 Limitations and Future work

The most substantial component of the CPU-Free Model that is yet to be implemented in DaCe is thread block optimization (sec. 3.1.3), as neither the DaCe scheduler nor the Python frontend exposes interfaces to address thread blocks individually or in groups. Future work will draft new syntax and Map types to allow such scheduling to be described in code. As a side effect of this, currently, our communication library nodes only schedule calls in either one CUDA thread followed by a global sync, or parallel maps using all available threads. As such, the level of communication overlap in the generated code is limited, and cooperative NVSHMEM calls such as `nvshmemx_putmem_block()` that collectively use an entire thread block to transfer data are not supported fully.

Another work we would like to explore in the future is generating more systematic persistent kernels, as the current persistent fusion transformation in DaCe rather naively merges a given subgraph into a single kernel. We predict that a smarter transformation can claim additional performance by taking device resource availability such as SM count, cache, and shared memory size into consideration.

Chapter 6

EVALUATION

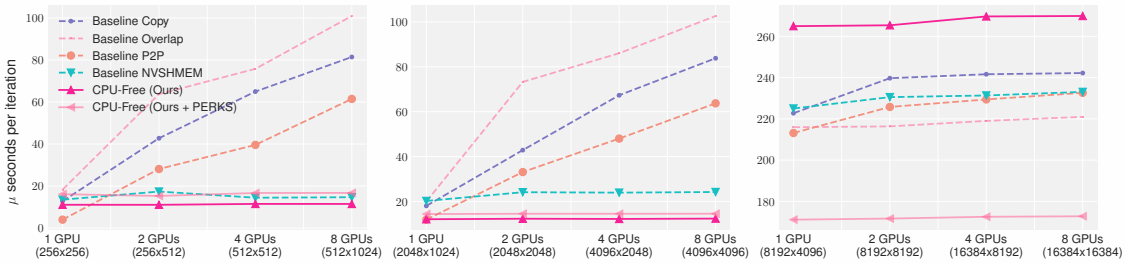


Figure 6.1: Weak scaling of 2D Jacobi stencil method with small to large domain sizes on up to 8 NVIDIA A100 GPUs

This chapter presents the performance scaling of the CPU-Free model compared to CPU-controlled baselines for 2D/3D Jacobi Stencil methods. The experiments were conducted on NVIDIA HGX machines with 8 NVIDIA A100 GPUs connected all-to-all through NVLink. The CUDA toolkit version is 11.8 with driver version 496.29.05 and the NVSHMEM library version 2.7.0 with OpenMPI 4.1.4. We report the minimum of 5 consecutive runs for each experiment.

Our speedup numbers are calculated and reported according to the following formula:

$$Speedup\% = \frac{T_{baseline} - T_{ours}}{T_{baseline}} \times 100\%$$

where T_x is the execution time of a version in seconds.

6.1 Handwritten Stencil Benchmarks

6.1.1 Experiment Setup and Code Variants

We categorize the experiments into three groups of domain sizes: small, medium, and large. The domain sizes are categorized as such based on device saturation: a small domain has fewer elements than needed to keep the entirety of the device busy, while the medium version has sufficient elements to utilize all thread blocks, and large domains over-saturate the device. For a 2D5pt Stencil on an NVIDIA A100 with 108 SMs, we select our domains as 256^2, 2048^2, and 8192^2, respectively.

The baselines are taken from NVIDIA's multi-gpu programming models repository [NVIDIA, 2022a], which contains Jacobi kernels with different communication schemes. We compare our implementations to the four best-performing versions:

- **Baseline Copy**: Standard CPU-controlled implementation with no explicit boundary overlap. This version only overlaps memory transfers with kernel execution using host-side asynchronous cudaMemcpy calls.

- **Baseline Overlap**: Same as above, but computes boundary rows in separate streams independently of the inner domain for explicit overlap, and synchronizes using host-side events. The explicit overlap this version performs is identical to our implementation.

- **Baseline P2P**: Communication is done through device-side direct load and stores in peer-to-peer memory. Although the communication is GPU-initiated in this version, synchronization is handled by the host.

- **Baseline NVSHMEM**: Uses device-side NVSHMEM calls for communication. This version utilizes the same family of NVSHMEM memory operations thatvie we use in our CPU-Free implementation, except in CPU-controlled discrete kernels. It additionally uses a dedicated kernel to synchronize neighboring GPUs to avoid redundantly synchronizing all processing elements. Both kernels are launched by the CPU in every time step.

- **CPU-Free**: Our implementation as described in Section 4.

We additionally measure multi-GPU PERKS performance with our communication scheme, as it tiles the compute kernel to overcome the limitations of the Cooperative Groups API. Lastly, we implement a 3D7pt stencil partitioned across the z axis, and adapt existing 2D baselines to 3D.

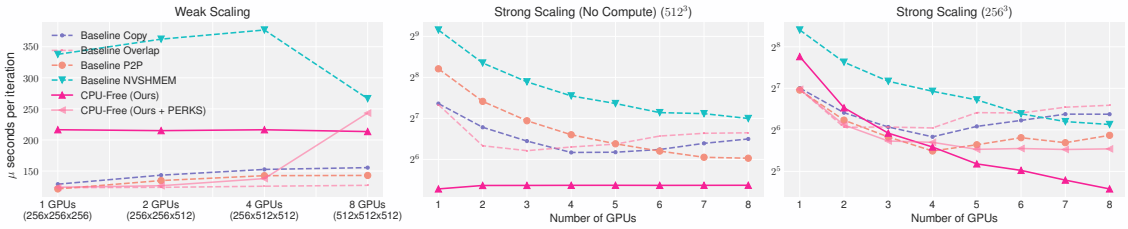


Figure 6.2: 3D Jacobi stencil weak and strong scaling experiments on up to 8 NVIDIA A100 GPUs

6.1.2 Scaling Experiments

Figure 6.1 shows weak-scaling measurements of per-iteration time in the aforementioned domain sizes. We vary the domain across all axes in alternating order as we double the number of devices for our weak scaling study. At 8 GPUs, we achieve speedups of 41.6% and 48.2% in small and medium domains respectively over the best-performing baseline (Baseline NVSHMEM), and 96.2% and 95.7% speedup over the fully CPU-Controlled Baseline Copy Overlap.

It should be noted that the performance degradation we experience in the largest domains compared to the baselines is due to subpar tiling in the computational kernels. As per the limitations set by the cooperative groups, kernels can only be launched in configurations that guarantee the co-residency of blocks, meaning, in our case, we only allocate one block of 1024 threads on each SM. In order to process the entire domain, we tile threads in software, which causes inefficiencies in large domains. We display multi-GPU PERKS results as an alternative, as their computational kernel provides better tiling. PERKS kernel with our communication

scheme achieves good weak scaling within a 9% dropoff at 8 GPUs, and 18.8% speedup over the baseline on the largest domain.

Figure 6.2 shows 3D Jacobi performance. Though the weak scaling displays lower overall performance due to the large domain, we measure better no-compute time, which is shown in the same figure in the middle at its largest domain.

Figure 6.2, also conducts strong scaling experiments on a constant large 3D domain to demonstrate synchronization overheads and the overlap amount as the number of devices increases. We notice that the CPU-Free version stays largely flat, while the CPU-controlled baselines degrade. With a small number of GPUs, each GPU has a large domain that is limited by computation, not communication. However, as the GPU count increases, communication and overheads become dominant, and CPU-free clearly exhibits its communication advantages, as seen in Strong Scaling (No Compute) experiments.

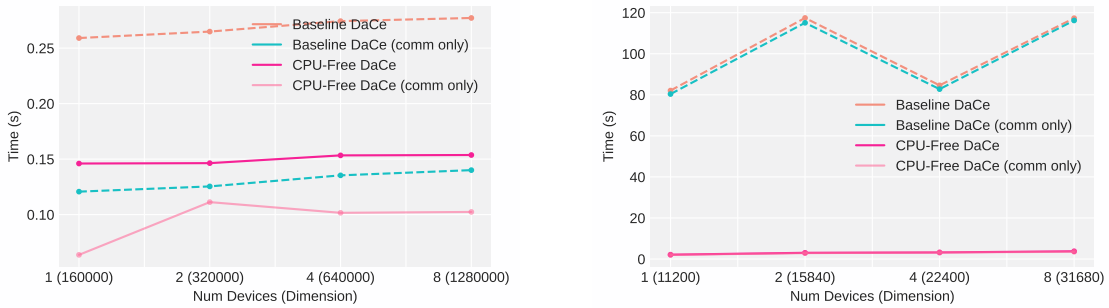
6.2 Compiler Generated CPU-Free Code with DaCe

6.2.1 Experiment Setup

We again adapt distributed stencil benchmarks (1D and 2D) from [Ziogas et al., 2021] as our baselines and convert them to CPU-Free execution. Although the benchmarks were originally presented for CPU nodes, we trivially port them to CUDA for fair comparison through the `GPUTransform` transformation in DaCe. We additionally apply an auto optimizer pass alongside the included `MapFusion` transformation.

We construct our CPU-Free program by applying a `GPUPersistentKernel` on the baseline code as an additional step and replace MPI calls with NVSHMEM in accordance with the discussion in Chapter 5. Namely, `Send` calls are replaced with signaled `Putmem*`, and `Recv` calls are replaced with `SignalWait*` nodes. We additionally omit global MPI barriers such as `Waitall` in favor of more granular flag-based synchronization already provided by the aforementioned NVSHMEM calls. No further changes are made to the program structure, execution order, or communication patterns expressed in the frontend.

We note that the codes in this section were compiled with CUDA toolkit version 12.2 and NVSHMEM 2.9.0. Our branch is synchronized to DaCe master commit c432824 and run with Python version 3.10.8.



(a) 1D Jacobi single element comm

(b) 2D Jacobi strided comm

Figure 6.3: Comparison of discrete distributed DaCe versus CPU-Free communication on up to 8 NVIDIA A100 GPUs

6.2.2 Experiment Variants

The applications we demonstrate exhibit two different communication schemes: single element put/send to each PE in case of 1D Jacobi 6.3a, and strided access in 2D 6.3b. Namely, the versions demonstrated are as follows:

- **Baseline Jacobi 1D**: CPU-controlled distributed 1D Stencil DaCe baseline. Each rank communicates a single element to two neighbors. The generated communication code schedules all `MPI` calls asynchronously and synchronizes with CUDA streams for each call before kernel launch. Host-side device-to-device `cudaMemcpy()` calls are generated to copy to global from temporary buffers.

- **CPU-Free Jacobi 1D**: CPU-Free implementation of the above using persistent fusion and NVSHMEM communication. All operations are done on the device side.

- **Baseline Jacobi 2D**: Partitions the domain as a grid, giving each rank four neighbors to communicate with. Schedules MPI calls similarly to Baseline Jacobi 1D. `MPI_Type_vector` is used to communicate strided data.

- **CPU-Free Jacobi 2D**: CPU-Free implementation of the above. We use `nvshmem_<TYPENAME>_iput()` and `nvshmem_signal_op()` for strided access and signaling.

6.2.3 Scaling Experiments

Figure 6.3 shows our weak scaling experiments of both applications. We begin with the largest domain sizes included in the original benchmarks [Ziogas et al., 2021] and scale them up to saturate the device. Similar the experiments in Section 6.1.2, we increase the domain size as we double the number of devices, and report the minimum execution time of 5 runs.

Jacobi 1D

At 8 GPUs, we see a performance improvement of 44.5% over the baseline in full execution time and 26.8% in communication latency 6.3a. Since each device communicates two elements regardless of the domain size in this application, we attribute most of the gains here to synchronization overheads.

Jacobi 2D

The improvements are more pronounced in this application where the amount of communication is greater as each device communicates with four neighbors, two with strided accesses. We note that the baseline is almost completely dominated by communication despite the large domain size, which takes up over 99% of the execution time. We observe a performance improvement of 96.8% at 8 GPUs and a weak scaling efficiency of 81.2% as shown in 6.3b despite the inefficiencies discussed above.

It should be noted that the pattern of increased execution time presenting at 2 and 8 GPUs in the baseline is likely due to unbalanced partitioning when the

number of devices is not a multiple of 4, giving us a rectangular split. We do not observe such inefficiency in the CPU-Free version.

Following our findings in section 6.1, we again see that CPU-Free execution is very effective in reducing communication latency. Notably, in the case of Jacobi 2D exhibiting a large amount of communication *and* non-contiguous memory access, we see massive improvements over the baseline by eliminating costly host involvement.

6.3 Overview

In summary, CPU-free may not always be optimal, but it has proven to be highly effective for small to medium domains when CPU-induced latencies consume a substantial portion of the runtime and communication/computation can be sufficiently overlapped. Our approach excels in strong scaling scenarios where the overhead ratio increases with GPU count. Conversely, traditional CPU-controlled implementations, both hand-tuned and generated with DaCe, fail to achieve sufficient overlap, causing communication and computation to be serialized in small-medium domains.

Chapter 7

RELATED WORK

7.1 Related Work

A number of works have provided and explored the components and building blocks used in this work to build the CPU-Free Model. Persistent kernels, the first technique for more autonomy that we discussed was originally presented in the work by [Gupta et al., 2012]. There have several been works utilizing persistent execution such as [Chu et al., 2019] that implemented a persistent GPU-based key-value store for increased throughput. PERKS [Zhang et al., 2022] takes advantage of persistent registers and shared memory in a single GPU to cache intermediate results and demonstrates significant performance improvements in Stencil and Conjugate Gradient applications.

The foundation for Thread Block specialization was initially laid in the work by [Tzeng et al., 2010] and later the Singe DSL by [Bauer et al., 2014b] that employed *warp* specialization for irregular computations. Together with persistent kernels, later work by [Chen et al., 2023] explores cooperative scheduling of both warps and blocks for irregular applications to build a task-parallel framework. Work by [Steinberger et al., 2014], WhippleTree, similarly applies persistent kernels to irregular workloads for graphical applications and features work queues of varying work item sizes i.e., warps and blocks. Groute [Ben-Nun et al., 2017] is another work that proposes a runtime for irregular workloads and implements asynchronous GPU work queues. Juggler [Belviranli et al., 2018] similarly uses both persistent kernels and a more general form of thread block specialization to implement a task-based execution model that treats SMs in a device as standalone processing units.

There have been a number of works towards GPU-initiated communication. Agostini et. al [Agostini et al., 2017, Agostini et al., 2018] explore GPUDirect

Async which allows devices to trigger and sync CPU-enqueued network transfers. Though not as sophisticated as current methods, this work is the first, to our knowledge, to grant GPUs power over the control path. [LeBeane et al., 2017] employs a similar mechanism of triggering network transfers, but implements a NIC hardware bypass to further eliminate CPU involvement.

NVSHMEM has recently taken the efforts further by providing a comprehensive API for fine-grained GPU-initiated communication with the PGAS model [NVIDIA, 2022b], and a number of works have explored and assessed it for various applications [Chen et al., 2022, Potluri et al., 2015, Potluri et al., 2017, Potluri et al., 2018, Hsu et al., 2020]. NVSHMEM has also seen adoption in several libraries and frameworks, such as Kokkos as a specialization of their Remote Spaces API [Ciesko, 2020], PETSc in their PetscSF communication component along with a communication programming model [Zhang et al., 2021], and LBANN to implement spatial-parallel convolution [Naoya Maruyama et al., 2020].

Chapter 8

CONCLUSION

In this work, we first presented a novel execution model for multi-GPU applications that eliminates host control, giving devices full autonomy. We achieve this by systematically combining several techniques such as persistent kernels, thread block specialization, device-side barriers and synchronization, and device-initiated communication. We implement *CPU-Free* 2D and 3D Stencil benchmarks and conduct weak and strong scaling studies in several domain sizes with 8 GPUs, and observe up to 96% speedups over *CPU-Controlled* baselines of varying degrees of host control.

In the second part of the thesis, we extended the work to a high-level parallel Python framework, DaCe, to automatically generate CPU-Free code. We then implemented Stencil benchmarks and conducted scaling experiments, similarly observing significant performance improvements over CPU-controlled code. Finally, we compared code generated with existing distributed communication facilities in DaCe, and discussed the future direction of the work.

BIBLIOGRAPHY

[Afanasyev et al., 2021] Afanasyev, A., Bianco, M., Mosimann, L., Osuna, C., Thaler, F., Vogt, H., Fuhrer, O., VandeVondele, J., and Schulthess, T. C. (2021). Gridtools: A framework for portable weather and climate applications. *SoftwareX*, 15:100707.

[Agostini et al., 2017] Agostini, E., Rossetti, D., and Potluri, S. (2017). Offloading communication control logic in gpu accelerated applications. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, page 248–257, New York, NY, USA. Institute for Electrical and Electronics Engineers.

[Agostini et al., 2018] Agostini, E., Rossetti, D., and Potluri, S. (2018). Gpudirect async: Exploring gpu synchronous communication techniques for infiniband clusters. *Journal of Parallel and Distributed Computing*, 114:28–45.

[Bauer et al., 2014a] Bauer, M., Treichler, S., and Aiken, A. (2014a). Singe: Leveraging warp specialization for high performance on gpus. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, page 119–130, New York, NY, USA. Association for Computing Machinery.

[Bauer et al., 2014b] Bauer, M., Treichler, S., and Aiken, A. (2014b). Singe: Leveraging warp specialization for high performance on gpus. *SIGPLAN Not.*, 49(8):119–130.

[Belviranli et al., 2018] Belviranli, M. E., Lee, S., Vetter, J. S., and Bhuyan, L. N. (2018). Juggler: A dependence-aware task-based execution framework for gpus. *SIGPLAN Not.*, 53(1):54–67.

[Ben-Nun et al., 2019] Ben-Nun, T., de Fine Licht, J., Ziogas, A. N., Schneider, T., and Hoefler, T. (2019). 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*, SC '19.

[Ben-Nun et al., 2017] Ben-Nun, T., Sutton, M., Pai, S., and Pingali, K. (2017). Groute: An asynchronous multi-gpu programming model for irregular computations. *SIGPLAN Not.*, 52(8):235–248.

[Chen et al., 2023] Chen, Y., Brock, B., Porumbescu, S., Buluc, A., Yelick, K., and Owens, J. (2023). Atos: A task-parallel gpu scheduler for graph analytics. In *Proceedings of the 51st International Conference on Parallel Processing*, ICPP '22, New York, NY, USA. Association for Computing Machinery.

[Chen et al., 2022] Chen, Y., Brock, B., Porumbescu, S., Buluç, A., Yelick, K., and Owens, J. D. (2022). 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*, SC '22, New York, NY, USA. Institute for Electrical and Electronics Engineers.

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

[Ciesko, 2020] Ciesko, J. (2020). Distributed memory programming and multi-gpu support with kokkos.

[Foley and Danskin, 2017] Foley, D. and Danskin, J. (2017). Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17.

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

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

[Harris et al., 2020] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

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

[LeBeane et al., 2017] LeBeane, M., Hamidouche, K., Benton, B., Breternitz, M., Reinhardt, S. K., and John, L. K. (2017). Gpu triggered networking for intra-kernel communications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA. Association for Computing Machinery.

[Meneghin et al., 2022] Meneghin, M., Mahmoud, A. H., Jayaraman, P. K., and Morris, N. J. W. (2022). Neon: A multi-gpu programming model for grid-based

computations. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 817–827.

[Meuer et al., 2023] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H., and Meuer, M. (2023). Top 500. `https://www.top500.org/`. Accessed: 2023-01-30.

[Naoya Maruyama et al., 2020] Naoya Maruyama, B. V. E., Ciesko, J., Wilke, J., Trott, C., Hsu, C.-H., Imam, N., Dinan, J., Langer, A., Newburn, C., and Potluri, S. (2020). Scaling scientific computing with nvshmem.

[NVIDIA, 2011a] NVIDIA (2011a). Cuda 4.0 release notes.

[NVIDIA, 2011b] NVIDIA (2011b). Cuda 9.0 release notes.

[NVIDIA, 2022a] NVIDIA (2022a). Multi gpu programming models. `https://github.com/NVIDIA/multi-gpu-programming-models`.

[NVIDIA, 2022b] NVIDIA (2022b). Nvidia openshmem library (nvshmem) documentation.

[NVIDIA et al., 2020] NVIDIA, Vingelmann, P., and Fitzek, F. H. (2020). Cuda, release: 10.2.89.

[Poole et al., 2011] Poole, S. W., Hernandez, O., Kuehn, J. A., Shipman, G. M., Curtis, A., and Feind, K. (2011). *OpenSHMEM - Toward a Unified RMA Model*, pages 1379–1391. Springer US, Boston, MA.

[Potluri et al., 2017] Potluri, S., Goswami, A., Rossetti, D., Newburn, C., Venkata, M. G., and Imam, N. (2017). 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)*, pages 253–262, New York, NY, USA. Institute for Electrical and Electronics Engineers.

[Potluri et al., 2018] Potluri, S., Goswami, A., Venkata, M. G., and Imam, N. (2018). Efficient breadth first search on multi-gpu systems using gpu-centric openshmem. In Gorentla Venkata, M., Imam, N., and Pophale, S., editors, *OpenSHMEM and Related Technologies. Big Compute and Big Data Convergence*, pages 82–96, Cham. Springer International Publishing.

[Potluri et al., 2015] Potluri, S., Rossetti, D., Becker, D., Poole, D., Gorentla Venkata, M., Hernandez, O., Shamis, P., Lopez, M. G., Baker, M., and Poole, W. (2015). 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*, OpenSHMEM 2015, page 18–35, Berlin, Heidelberg. Springer-Verlag.

[Shimokawabe et al., 2011] Shimokawabe, T., Aoki, T., Takaki, T., Endo, T., Yamanaka, A., Maruyama, N., Nukada, A., and Matsuoka, S. (2011). Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA. Association for Computing Machinery.

[SPCL, 2023] SPCL (2023). Dace documentation.

[Spotz and Carey, 1995] Spotz, W. and Carey, G. (1995). High-order compact finite difference methods. In *Preliminary Proceedings in International Conference on Spectral and High Order Methods*, pages 397–408, Houston, TX, USA. International Conference on Spectral and High Order Methods.

[Steinberger et al., 2014] Steinberger, M., Kenzel, M., Boechat, P., Kerbl, B., Dokter, M., and Schmalstieg, D. (2014). Whippletree: Task-based scheduling of dynamic workloads on the gpu. *ACM Trans. Graph.*, 33(6).

[Strikwerda, 2004] Strikwerda, J. C. (2004). *Finite Difference Schemes and Partial Differential Equations, Second Edition.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[Tzeng et al., 2010] Tzeng, S., Patney, A., and Owens, J. D. (2010). Task management for irregular-parallel workloads on the gpu. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, page 29–37, Goslar, DEU. Eurographics Association.

[Wahib and Maruyama, 2014] Wahib, M. and Maruyama, N. (2014). Scalable kernel fusion for memory-bound GPU applications. In *SC*, pages 191–202, New Orleans, LA, USA. IEEE Computer Society.

[Yamaguchi et al., 2017] Yamaguchi, T., Fujita, K., Ichimura, T., Hori, T., Hori, M., and Wijerathne, L. (2017). Fast finite element analysis method using multiple gpus for crustal deformation and its application to stochastic inversion analysis with geometry uncertainty. In *ICCS*, volume 108 of *Procedia Computer Science*, pages 765–775, Zürich, Switzerland. Elsevier.

[Zhang et al., 2021] Zhang, J., Brown, J., Balay, S., Faibussowitsch, J., Knepley, M., Marin, O., Mills, R. T., Munson, T., Smith, B. F., and Zampini, S. (2021). The petscsf scalable communication layer. *IEEE Transactions on Parallel and Distributed Systems*, 33(4).

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

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

[Ziogas et al., 2021] Ziogas, A. N., Schneider, T., Ben-Nun, T., Calotoiu, A., De Matteis, T., de Fine Licht, J., Lavarini, L., and Hoefler, T. (2021). Productivity, portability, performance: Data-centric python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA. Association for Computing Machinery.