Harmonic CUDA: Asynchronous Programming on GPUs

Jonathan Wapman
University of California, Davis
Davis, California, USA
jdwapman@ucdavis.edu

Sean Treichler
NVIDIA
Santa Clara, California, USA
sean@nvidia.com

Serban D. Porumbescu
University of California, Davis
Davis, California, USA
sdporumbescu@ucdavis.edu

John D. Owens
University of California, Davis
Davis, California, USA
jowens@ucdavis.edu

Abstract

We introduce Harmonic CUDA, a dataflow programming model for GPUs that allows programmers to describe algorithms as a dependency graph of producers and consumers where data flows continuously through the graph for the duration of the kernel. This makes it easier for programmers to exploit asynchrony, warp specialization, and hardware acceleration. Using Harmonic CUDA, we implement two example applications: Matrix Multiplication and GraphSage. The matrix multiplication kernel demonstrates how a key kernel can break down into more granular building blocks, with results that show a geomean average of 80% of cuBLAS performance, and up to 92% when omitting small matrices, as well as an analysis of how to improve performance in the future. GraphSage shows how asynchrony and warp specialization can provide significant performance improvements by reusing the same building blocks as the matrix multiplication kernel. We show performance improvements of 34% by changing to a warp-specialized version compared to a bulk-synchronous implementation. This paper evaluates the strengths and weaknesses of Harmonic CUDA based on these test cases and suggests future work to improve the programming model.

CCS Concepts: • **Computing methodologies → Parallel programming languages**.

Keywords: GPU, CUDA, programming model, asynchronous, GEMM, GraphSage

ACM Reference Format:
Jonathan Wapman, Sean Treichler, Serban D. Porumbescu, and John D. Owens. 2023. Harmonic CUDA: Asynchronous Programming on GPUs. In *International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '23), February 25–March 1, 2023, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3582514.3582517

1 Introduction

Modern GPUs are more than just a group of thread processors. Over time, GPUs have added specialized hardware units such as Tensor Cores of many varieties for machine learning and dense linear algebra, Ray Tracing cores for realistic rendering, Direct Memory Access (DMA), and Tensor Memory Accelerator (TMA) units for asynchronous memory movements between off-chip and on-chip memory, and Transformer Engines for machine learning, with potentially more to come [4, 6, 15, 19]. However, taking advantage of these specialized units typically requires major code rewrites, and orchestrating data movement in a performant way often requires skilled CUDA programming ability. The need to rewrite software for each new GPU architecture is a major barrier to the adoption of new hardware features. NVIDIA may be able to allocate these resources to commonly used and performance-critical libraries that automatically bring new hardware features to end-users, but this does not apply to custom code written by the users themselves. At the same time, GPU programming models have become increasingly asynchronous (Section 2) and it is desirable to overlap computation with memory transfers or differing types of computation depending on available accelerator features.

At a higher level of abstraction, NVIDIA and third parties provide many libraries that implement critical GPU kernels such as matrix multiplication [14, 20], block-wide or device-wide collective operations such as prefix sums or reductions [11, 17], or more complex algorithms such as graph algorithms [8]. These libraries provide highly optimized implementations of these kernels, but they lack flexibility. For example, a library such as CUTLASS [14] provides a matrix multiplication kernel that utilizes the entire device and building blocks for implementing custom device-wide kernels but does not support the warp-centric configuration necessary for an application like GraphSage (Section 5). This inflexibility makes it difficult to take advantage of the highly optimized kernels provided by these libraries, and instead, forces programmers to write their own kernels from scratch.

Additionally, CUDA libraries typically do not have the flexibility to perform an operation at any level of the GPU's compute hierarchy. A GPU programmer may want to perform an operation with a single thread, warp, or block, a subset of a block, an entire grid, or a subset of a grid. However, CUDA is not well-suited to elegantly expressing operations where the core *algorithm* is the same even though the *location* of the data and the assigned compute group may change.

To solve these challenges, we present *Harmonic CUDA*: a programming model for asynchronous producer/consumer computation on modern GPUs that enables programmers to *describe* the dataflow of their code, while relying on highly optimized backends to handle scheduling, synchronization, hardware acceleration, and storage management. The primary goals of Harmonic CUDA are:

- **Computation/Location Abstraction:** Programmers should be able to express the *what* of their computations without worrying about the *where* or *when*.
- **Performance:** Programmers should be able to rely on Harmonic CUDA to use best-available implementations for its backend, which may include highly optimized software libraries, architecture-specific accelerators such as Tensor Cores, or future hardware and software features as they become available.
- **Composability:** The programmer should be able to construct a Dataflow and treat that Dataflow as an individual Node within other Dataflows.
- **Programmability:** Harmonic CUDA should be an intuitive programming model that makes it *easier* for a programmer to think about their code in terms of data flow, and to write code that is easy to understand and maintain. This includes a simple, easy-to-learn programming model and API.
- **Harmony:** The programmer should be able to use Harmonic CUDA *alongside* traditional CUDA code.

We evaluate Harmonic CUDA on an implementation of a memory copy kernel in Harmonic CUDA that shows the Harmonic CUDA API and backend implementation (Section 3), and implementations of matrix multiplication (Section 4) and GraphSage (Section 5) that demonstrate Harmonic CUDA's performance, the benefits of warp specialization, and the benefits of the reuse of building blocks.

2 Related Works

Hardware Asynchrony. In recent years, GPUs have added more asynchronous functionality. NVIDIA's Ampere architecture includes asynchronous Direct Memory Access (DMA) units that use dedicated hardware units to copy sequential regions of memory directly from global memory to shared memory without the need for intermediate copies to thread registers [15]. NVIDIA's Hopper architecture extends this concept to a Tensor Memory Access unit, which performs the same function but for 2-dimensional tiles of a matrix [9, 19]. Additionally, there is a large body of research into domain-specific accelerators [12], which feature asynchronously running hardware units connected with intermediate buffers [23]. Harmonic CUDA addresses the programming challenge of efficiently targeting an increasing number of asynchronous by creating abstractions around the actual implementation of logical operations and the data movements between them.

GPU Software Asynchrony. CudaDMA divides a block on the GPU into DMA warps for performing memory transfers, and computation warps for performing any necessary computation [2]. Building on the concepts of CudaDMA, Warp Specialization [3] allows programmers to define multiple execution paths through kernels. Based on a runtime condition a warp can, for example, transfer data from global to shared memory or alternatively, perform computation on data in shared memory. All warps within the same block may continue to communicate with each other over shared memory and may take advantage of efficient synchronization mechanisms. Harmonic CUDA provides a more general approach to warp specialization that abstracts away the low-level details of warp specialization and allows programmers to use more flexible specialization configurations.

Libcu++ [22] provides abstractions such as pipelines and barriers to manage asynchronous GPU hardware. Harmonic CUDA leverages these primitives as building blocks to create a higher-level programming abstraction.

Several works [1, 13] aim to hide complexities of asynchronous GPU programming (such as synchronization, Direct Memory Accesses, and memory management) using a producer-consumer model, but these focus on kernel-level dataflows between the CPU and one or more GPUs. In contrast, Harmonic CUDA provides a programming model usable *within* a GPU kernel at runtime. Additionally, while Harmonic CUDA may target heterogeneous systems for future work, the programming model itself is fundamentally different, since it treats the CPU or multiple GPUs as just another user-specified compute location.

CPU Software Asynchrony. LabView and Simulink are node-based graphical dataflow programming models commonly used on the CPU [16, 18]. LabView and Simulink both provide libraries of Nodes that a programmer connects together in a dataflow to perform some computation, with typical examples being signal processing or control loops. Harmonic CUDA differs in that it is a text-based programming model that additionally provides for the concept of "where" to store and/or perform computation.

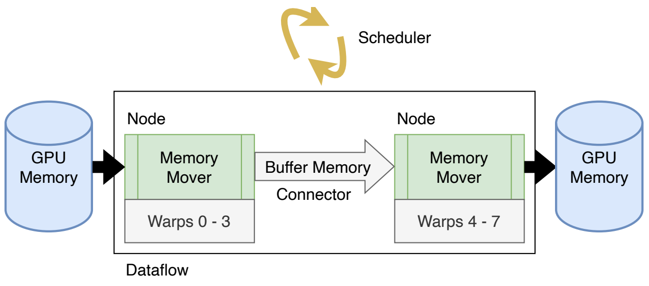

Figure 1. A Harmonic CUDA Memory Copy Dataflow. The producer and consumer map to an arbitrary compute location, and the buffers map to an arbitrary storage location. The scheduler takes advantage of hardware asynchrony while respecting resource limitations.

StreamIt [25] is a text-based DSL for programming streaming applications, but this model focuses primarily on signal processing, rather than on general-purpose parallel computing. It also does not include any concept of "where" to store and/or perform computation beyond offering support for CPU multi-threading.

In the Senders programming model [7], a "sender" object contains a computation that runs asynchronously. The sender returns a single item (such as an int, pointer, struct, etc.) before a dependent sender begins. Although it is an asynchronous programming model, the Senders model is not a true producer/consumer dataflow pipeline where data flows continuously. Harmonic CUDA is compatible with the Senders model. For example, programmers can use Harmonic CUDA to more easily write a CUDA kernel that they can then launch as the computation of a Sender.

There are many examples of parallel programming and dataflow programming languages on both the GPU and the CPU [5]. In particular, Halide [24] similarly separates the logical computation from a schedule, but is not a dataflow programming model.

3 Programming Model

Figure 1 shows an example of a basic memory copy operation implemented as a Harmonic CUDA Dataflow using two instances of data transformation (a producer and a consumer) and an intermediate buffer. The basic unit of Harmonic CUDA is a *Node*, which is an abstraction of a program (Section 3.1). A Node embodies tasks such as "perform a prefix sum on the elements given an input," "perform elementwise additions on the two inputs," or "do a matrix multiplication given two input matrices." A Node may run on a thread, a block, a subset of a block, an entire grid, or a subset of a grid. It may have a software backend or a hardware-accelerated backend, and may coexist on the same hardware with many other Nodes. Using this abstraction, the Node defines *what* the user wants to do while giving the flexibility to easily change *where* the computation runs, and also gives the system the power to decide *when* it happens.

Connectors join Nodes and capture synchronization and storage management (Section 3.2). The user provides Connectors with input and output Nodes as well as the intermediate data storage location, while the Nodes and Connectors choose highly optimized backends based on the functionality of the Nodes, the size of the data chunks transferred between Nodes, the locations of the Nodes, and the locations of the intermediate data storage.

A *Dataflow* collects Nodes and Connectors into a single unit of computation and assists in scheduling of Nodes onto hardware (Section 3.3). Furthermore, Dataflows are composable—that is, they themselves can be used as a Node within a larger Dataflow.

Finally, Harmonic CUDA provides an automatic Node scheduler that is able to efficiently schedule Nodes onto hardware for many common cases, and APIs that the programmer can use to manually schedule Nodes onto hardware for more complex cases (Section 3.5).

To enable Harmonic CUDA to coexist *alongside* traditional CUDA, threads can interact with aforementioned components. For example, Harmonic CUDA could provide Nodes whose function is to read data from a thread-provided location, or to fill data into a location where other GPU threads can consume the data directly.

Workflow. Programmers should use Harmonic CUDA in a three-step process. First, they should conceptualize their algorithm as a dependency graph of producers and consumers without considering the physical mapping of computations or storage locations to the GPU. Next, they must instantiate the dataflow in code and decide where to map the Nodes and Connectors to the GPU. Here, simply supplying the correct flags is enough for the system to take care of the backend. Finally, they should profile their kernel and adjust the mapping based on results, such as separating two Nodes to make them warp-specialized or using shared memory instead of registers.

3.1 Nodes

A Node is an abstraction of data transformation and movement that specifies *what* computation is being done. *Where* and *when* are instead specified by a programmer-provided compute location parameter and by Harmonic CUDA's scheduler. Nodes utilize NVIDIA's "Cooperative Groups" API [11] to specify which hardware unit(s) to use. The actual implementation of a Node, or its *Backend*, may be highly optimized third-party CUDA backends, hardware accelerators, or custom code written by the end user that builds on top of the Node framework. The compiler can often determine the specific backend of a Node at compile time based on the locations of the input and output data of the Node, the compute location the Node runs on, and the hardware capabilities of

```
__global__ memcpy_kernel(int *global_data, int size) {
  __shared__ int shared_data[...];
  auto Global2Shared = make_MemMover(/*compute_group=*/GridGroup,
                                     /*input_location=*/LocationGlobal,
                                     /*output_location=*/LocationShared,
                                     /*input_data=*/global_data,
                                     /*output_data=*/shared_data,
                                     /*size=*/size * sizeof(int));
  auto Shared2Global = make_MemMover(/*...*/);
  // ...
}
```

Listing 1. Memory Copy Node Instantiation.

the GPU architecture. Other factors, such as runtime flags, may also determine the backend of a Node.

To support the programmability goal, Harmonic CUDA provides Node templates that programmers can customize to their needs, either by building on top of a template or by providing a concise lambda function that implements the Node's behavior. This allows programmers to easily create Nodes that are highly optimized for their specific use case, while still utilizing the high-level programming model.

The following sections describe several "perspectives" of Nodes: what the programmer cares about, what the backend implementer needs to know, and what the scheduler needs.

Node User. A Node user must understand at a high level what a Node does and what parameters to change to control its behavior. Given a Node, the programmer is responsible for assigning the Node to a Compute Location based on an algorithmic design decision. The programmer is also responsible for providing a Node with the correct inputs and outputs at all relevant ports. To run the Node, the programmer should provide the desired batch size or total amount of data. This can either be at runtime or compile time. Each Node includes a `progress()` method, which instructs the Node to make forward progress given its available input data and output space. The programmer must understand what action a given Node's `progress()` method takes, and must also understand what a Node requires to be "ready" or "finished."

Listing 1 shows an example of how to instantiate two "MemMover" Nodes for a basic Memory Copy example. Both Nodes map to a `GridGroup` and internally calculate indices relative to other blocks. Alternatively, the user could assign each Node to a `BlockGroup` and use Harmonic CUDA helpers to calculate global memory pointers and copy sizes per-block. The `LocationGlobal` and `LocationShared` parameters allow Harmonic CUDA to make compile-time optimizations. Variations of `make_MemMover` (provided by Harmonic CUDA) could provide greater runtime flexibility but higher runtime cost by omitting the need to specify input and output data locations at compile time.

Node Implementer. It is the Node implementer's responsibility to provide the functionality of the Node given the input and output locations of the data, the target GPU architecture, the runtime flags of the Node, or the compute location the Node is mapped to. As part of the implementation, the Node must be able to query input and output Connectors to determine how much data is available and where to read from or write to. Internally, a Node must implement `is_finished()` and `is_ready()` methods that are for dynamic scheduling, along with any necessary internal state tracking based on the total amount of data or the batch size. Finally, Nodes must perform all data movement and transformation when calling the `progress()` method, including any internal state updates. It is possible to implement several variants of the `progress()` method, such as supplying pipelines or other synchronization variables as needed depending on the configuration of the overall Dataflow.

```
__device__ progress(int buffer_size, PipelineT &pipe) {
#if __CUDA_ARCH__ >= 800
  // Special asynchronous copy for Ampere GPUs using DMA units.
  cuda::memcpy_async(
      this->get_compute_group(), this->output_base + this->output_offsett,
     this->input_base + this->input_offset, buffer_size * sizeof(int), pipe);
#else
  // For non-Ampere GPUs...
#endif
  internal_state_update(buffer_size);
}

__device__ bool is_finished() { return elements_copied >= max_copies; }
__device__ bool is_ready(/*...*/) {
  /* ... */

  return !(input_elems_avail == 0 || output_space_avail == 0 ||
      elements_copied >= max_copies);
}
```

Listing 2. Architecture-Specialized Node Backend.

Listing 2 shows an example of the MemMover backend from Listing 1, where we demonstrate how for supported hardware such as Ampere GPUs, the Node can use the DMA engine for asynchronous memory copies. This reduces register usage and allows threads to do other work while copies complete.

Node Scheduler. From the scheduler's perspective, the functionality of a Node is irrelevant. The scheduler only needs to know if a Node is ready to run, if it is finished, and how much data it can process in a single call to `progress()`. If the scheduler knows the Node's batch size at compile time, it can generate a static schedule for the Node, which has much less overhead than a runtime scheduler. For the dynamic scheduler, `is_finished()` and `is_ready()` methods allow the scheduler to make runtime decisions about which Node to run next.

3.2 Connectors

Connectors are responsible for managing the intermediate buffer storage between Nodes. A Connector's backing storage may be global memory, a shared memory buffer, thread-local registers, or other specialized locations such as Tensor Core registers. Connectors must manage synchronization between Nodes, handle optimizations such as double buffering, and assist the scheduler by negotiating the amount of data

```
__shared__ ConnectorSharedState memcpy_connector_shared_state;
auto memcpy_connector = make_connector(Global2Shared, Shared2Global,
                                       &memcpy_connector_shared_state);
```

Listing 3. Connector Instantiation.

that a single call to a Node's `progress()` method processes. In many ways, a Connector is similar to a *Buffet* [23], in that it has a fixed capacity, both queue-like and array-like operations, and forms the basis of connections in a dataflow graph.

Connector User. The programmer must provide the Connector information about how much storage it has to manage, how many buffers it has, its batch size, and provide a shared state for the Connector. Some parameters may be specified at compile time for better optimization, and others at run time.

Connector Implementation. The Connector assists its input and output Nodes with synchronization, pipelining or barrier operations as instructed by the scheduler. It must internally encapsulate any necessary pipelines or synchronization helpers and provide Nodes with a way to query available space/data as well as the read/write offsets.

Scheduler. Given two Nodes with user-specified compute locations, the Connector is responsible for choosing the appropriate synchronization between the Nodes. This may be a pipeline, a `__syncthreads()` barrier, or potentially a no-op depending on Node behavior and configuration. The scheduler must query the Connectors to determine the batch size for Node scheduling at either compile time or runtime.

3.3 Dataflows

Dataflows represent an encapsulation of a diagram of Nodes and Connectors. To build a Dataflow, programmers first instantiate a Node or Connector, and then add it to the Dataflow in a functional programming style, where they incrementally build up a Dataflow by chaining together Nodes and Connectors (Listings 1, 3, and 4). Depending on the configuration of Nodes and Connectors, the Dataflow may be able to do higher-level reasoning to optimize the application's performance, such as fusing Nodes together.

To support Harmonic CUDA's goal of composability, a programmer can pass a Dataflow as a Node to another Dataflow. Note that while our prototype implementation of Harmonic CUDA currently does not support this, the abstraction *does*, and it is a clear next step for future work, with possible challenges being how to support the dynamic scheduling of a Dataflow that is passed as a Node and how to handle cases where the programmer wants to map a complete Dataflow to a variety of compute locations.

```
auto BaseDF = make_dataflow();
auto [DF_Global2Shared, ID_Global2Shared] = BaseDF.add_node(Global2Shared);
auto [DF_Shared2Global, ID_Shared2Global] =
    DF_Global2Shared.add_node(Shared2Global);
auto [DF_Connected, ID_Connected] =
    DF_Shared2Global.add_connection(memcpy_connector);
```

Listing 4. Dataflow Instantiation.

3.4 Interaction with CUDA

Harmonic CUDA aims to coexist with traditional CUDA code. To achieve this, threads can fill data into an input buffer of a Dataflow, consume data from an output buffer of a Dataflow, or interact with Node scheduling directly. The Node framework should also enable programmers to pass in CUDA lambda functions as parameters to a Node. For example, an "Elementwise" Node can consume elements one at a time from each input buffer. The programmer can then define the functionality of the Node, such as elementwise addition, summation over the stream, or some other operation.

3.5 Scheduling

3.5.1 Asynchronous Scheduling. Figure 1 shows an example of two memory movement Nodes mapped to different halves of the same block (e.g., warps 0–3 of the reader and 4–7 of the writer) following the warp specialization approach in Section 2. The Nodes pass data between one another using a shared memory buffer abstracted by the Connector. To optimize, the Connector can use double buffering, allowing the reader and writer to work asynchronously and in parallel. The Connector handles synchronization between the two Nodes using the "pipeline" abstractions from libcu++.

3.5.2 Interleaved Scheduling. The interleaved schedule presented in Figure 2 is logically the same Dataflow as before. However, in the interleaved schedule, the *location* to where the Nodes and Connector map are different. Now, both Nodes map to the same Cooperative Group, and the intermediate storage location becomes registers rather than shared or global memory since threads do not need to share data between one another. The functional end result of the algorithm is the same, but not the implementation. Now, the two Nodes share the same compute resources through time slicing in a pattern of read, write, read, write, read, etc. Additionally, while the previous example would have been able to take advantage of Ampere's asynchronous global-to-shared memory transfers, the interleaved schedule does not. This is because the intermediate storage location is a register, rather than shared memory.

3.5.3 Bulk-Synchronous Scheduling. Algorithms often require barriers between stages, and although Harmonic CUDA is a continuous dataflow producer/consumer programming model, it also supports sequenced operations. For example, a kernel where the programmer wants to perform an in-place sort followed by an in-place scan, they can set

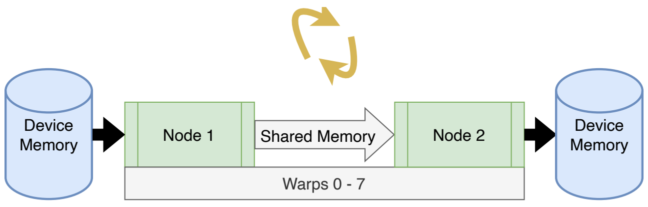

Figure 2. Interleaved Node Scheduling.

the chunk size equal to the data array size, blocking the scan Node until it has all available input data. Harmonic CUDA can also automatically determine if the sort is in-place or not by analyzing input and output storage addresses, in both asynchronous and interleaved schedules.

3.5.4 Manual Scheduling.

The Manual Scheduling API enables advanced users to schedule Nodes to run on GPU hardware. Programmers can query a Node's readiness, invoke its `progress()` method, and set termination conditions.

3.5.5 Automatic Scheduling.

From the user's perspective, the only automatic scheduling API is to call the `run()` method on a Dataflow. In many cases the automatic scheduler is able to generate a static schedule at compile time for some portions of the Dataflow. For example, in a GEMM example, the scheduler knows some tile sizes (such as the sizes for the Tensor Cores) at compile time based on the GPU architecture and the data type. In such cases, the automatic scheduler does not need to make each Node query its Connector for available space and can perform other optimizations such as unrolling. For schedules that cannot be determined at compile time, the automatic scheduler queries Nodes to determine if they are ready to run, have enough input data, and have enough output buffer space. The automatic scheduler is currently functional but not performant, making this a promising area of future work.

3.6 Summary

When programming with Harmonic CUDA, the programmer has several key design decisions and implementation tasks. First, how can the programmer break down their algorithm into Harmonic CUDA Nodes? In many cases, Harmonic CUDA has off-the-shelf Nodes a programmer can use. In other cases, the programmer may need to implement the desired functionality on top of Node templates in the Harmonic CUDA framework. Second, the programmer must connect all Nodes together into a Dataflow, specify and allocate the intermediate storage locations (future iterations of Harmonic CUDA should make this more automatic if possible), connect all Nodes together into a Dataflow, and specify desired optimizations such as double buffering. Finally, the programmer must determine which hardware units each Node maps to. This will likely call for some trial-and-error experimentation. In many cases the programmer may want to run all Nodes on the same cooperative group in the style of a traditional CUDA program, and in other cases the kernel may benefit from Cooperative Group Specialization.

An important abstraction of Harmonic CUDA is the distinction between a frontend user, whose job is to instantiate a graph of nodes that logically expresses their desired algorithm, and the backend implementer, who is responsible for creating performant implementations of individual nodes. The goal of Harmonic CUDA is that, if the frontend user fully understands Harmonic CUDA's programming model and the high-level functionality of a Node, then the backend developer can implement the Node without needing to understand the frontend user's code. Harmonic CUDA acts as an intermediate abstraction layer that hides the details of the hardware and the low-level programming model while still delivering high performance. Ideally, Harmonic CUDA should nearly match or even exceed the performance of a hand-written implementation. Harmonic CUDA intelligently chooses the appropriate backends or configuration parameters of each individual node based on the hardware and dataflow graph. Although future work is needed to evaluate this approach across a wide variety of algorithms, Harmonic CUDA is a promising approach to programming GPUs that is both expressive and performant in the general case without requiring significant programmer effort to generate highly optimized code. For more challenging Dataflows, Harmonic CUDA Nodes can expose configuration options that allow the programmer to manually tune some aspects of the implementation without needing to completely hand-write the program.

4 Matrix Multiplication

Generalized Matrix Multiplication (GEMM) is typically implemented as a device-wide operation. For example, cuBLAS, NVIDIA's closed-source matrix multiplication library, supports GEMM operations launched from the GPU or CPU that utilize the entire device. CUTLASS, NVIDIA's open-source matrix multiplication library, attempts to support more composability in how the GPU handles tile sizes for a variety of matrix shapes, but this is still primarily at the device level, where all threads, warps, or blocks in a GPU work together to solve a single GEMM problem. This limits the ability of programmers to experiment with GEMM use cases that do not require a full GPU or that combine GEMM building blocks with other operations (e.g., Section 5). GEMM also commonly utilizes hardware acceleration units such as DMA, TMA, or Tensor Core units.

As GPUs add more of these units over time, it is important for programmers to have abstractions around these logical operations so they do not need to do major code rewrites. To address this, our Harmonic CUDA GEMM implementation

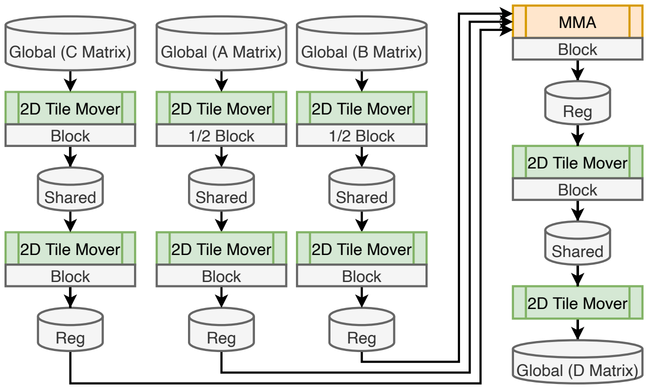

Figure 3. Matrix Multiplication using Harmonic CUDA 2D Tile Mover and MMA Nodes.

provides generic building blocks for memory movement and compute in a way that abstracts away the location, size, or composition of the building blocks. This allows us to utilize the same core building block no matter where our input and output data sources are. For example, moving a 2D tile of a matrix is logically the same operation whether it is a global-to-shared or a shared-to-register transfer, even though the implementation may change.

4.1 Implementation

In contrast to Harmonic CUDA, CUTLASS only supports a fixed set of operations at each level of the hierarchy. At the device level, this is the full GEMM operation; at the block level, a fixed-size tile of the matrix; and at the warp level, a smaller fixed-size tile that corresponds to the size of a tensor core, with separate implementations for data movement between each level. Harmonic CUDA allows the programmer to reason about the matrix multiplication building blocks at a higher level of abstraction, while allowing the backend to take care of the performance-critical optimizations that could be provided by CUTLASS as a lower-level backend.

GEMM is a hierarchical algorithm. We first tile the output matrix in global memory and iteratively read in tiles of the A and B matrices from global to shared memory. We then repeat this pattern, tiling a block-level output matrix for warps, and for threads, and so on. Logically, these are all simply 2D tile movement patterns between different levels of the memory hierarchy. In Harmonic CUDA we can express this as a single "2D Tile Mover" Node regardless of the size of the tile, where it runs on the GPU, or the location of its inputs or outputs. Finally, we include an "MMA" Node that performs the matrix multiplication and accumulation for a single tile of the output matrix. By abstracting away any optimization details common in Matrix Multiplication kernels, we enable programmers to think about GEMM as a small set of building blocks that move memory in 2D tiles between different levels of the memory hierarchy. We show the Harmonic CUDA Dataflow in Figure 3.

Although at a high level the only Node the programmer needs to think about is the "2D Tile Mover" Node, internally, the Node specializes its backend based on the input and output locations, which compute group it is assigned to, whether the data is row- or column-major format, the data type of the matrix, and other runtime and compile-time parameters. This specialization should be invisible to the user. Similarly, for the MMA Node we logically map the MMA Node to the entire block, where the inputs are registers. Although internally the MMA Node takes advantage of the GPU's Tensor Cores and complex logic for index mappings, this is again invisible to the user. The MMA Node could be alternatively mapped to a thread, a single warp, or a subset of a block and still *logically* perform the correct implementation.

For simplicity, we adapt NVIDIA's `dmmaTensorCoreGemm` example [21] as the backend for all Nodes, modifying operations to support arbitrary cooperative groups, but note that any valid GPU GEMM library (such as cuBLAS or CUTLASS) could form the backend of Harmonic CUDA's GEMM nodes. To match the implementation in the sample code repository, we map the 2D Tile Mover Node to separate halves of each block for the global-to-shared memory copies of A and B matrices, as this is more efficient than mapping each to the entire block due to the required shared memory tile sizes. However, the strength of Harmonic CUDA is that it is easy for the programmer to experiment with configurations of Nodes onto compute hardware; for example, mapping all Nodes to the same block would be equally valid (albeit slower).

We manually schedule Nodes to allow optimal kernel performance. We assume a more sophisticated automatic scheduler with minimal overhead could statically determine an appropriate schedule, as the compiler already knows the Tensor Core tile size at compile time. Note that the programmer does not need to specify the tile size, as Nodes and Connectors automatically select the implementation and size based on architecture capabilities, allocated resources, and the data types of the operation.

4.2 Results

We compare our results in Figure 4 against cuBLAS, NVIDIA's highly optimized, closed-source GEMM library. We conduct all experiments on an NVIDIA A100 GPU using CUDA Toolkit version 11.6. Matrices evaluated include square matrices (up to 32768 rows/columns), tiles that are an integer multiple of SMs to avoid workload imbalance, mixed aspect ratios, and (M, N, K) = (6912, 64, 4608), approximating the amount of work done in the GraphSage kernel (Chapter 5). Our tests show a geomean average speedup of 0.8X vs. cuBLAS, but excluding the 3 smallest matrices, where cuBLAS has specialized routines for small matrices, our geomean speedup improves to 0.92X.

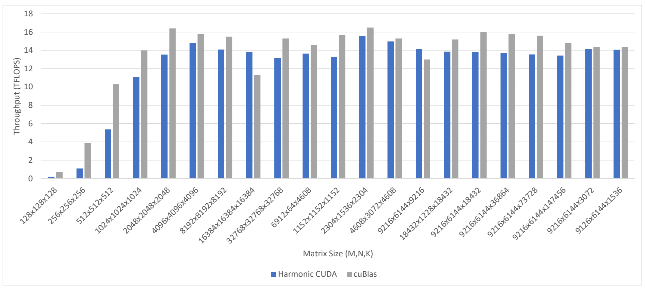

Figure 4. GEMM Performance Results.

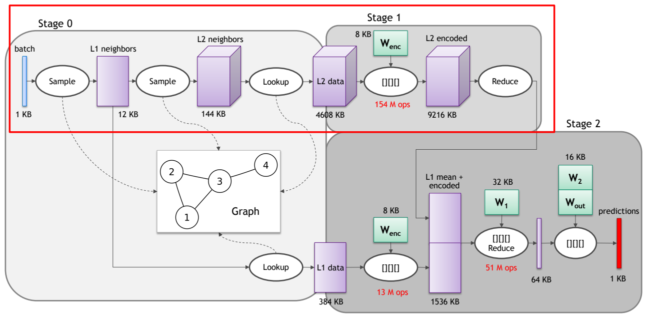

Figure 5. GraphSage forward pass with the subset implemented in Harmonic CUDA highlighted in red. Stage 0 represents sparse gathers and Stage 1 represents repeated GEMM operations. Reproduced with permission from Angshuman Parashar.

We believe that the performance discrepancy between Harmonic CUDA and cuBLAS is because although Harmonic CUDA currently takes advantage of the A100's Tensor Cores and DMA units, it lacks many of the low-level optimizations that cuBLAS performs such as pipelined memory movement, assembly-level optimizations, heuristics for tile sizes, and many others. As such, this performance discrepancy is expected. The goal of this experiment is not to *beat* cuBLAS, but rather to show that we are able to express matrix multiplication as a dataflow graph of producers and consumers with a moderately fast backend that allows straightforward implementation and experimentation. We fully expect that in the future, given more development time, an open-source library such as CUTLASS (which achieves performance parity with cuBLAS) could instead become the backend of Harmonic CUDA's GEMM Nodes and could incorporate many additional optimizations such as pipelining or dimension-specific variants. Note that the primary barrier to implementing these optimizations in Harmonic CUDA is engineering time, rather than limitations of the programming model itself. Pipelined memory movement fits perfectly into the "Connector" abstraction, assembly-level optimizations may form the backends of Nodes, and heuristics for tile sizes are a fundamental part of the "Node" abstraction, where the Node picks the appropriate backend implementation depending on the architecture, data type, matrix size, and other parameters. We believe that the Harmonic CUDA programming model is well-suited to these optimizations.

5 GraphSage

GraphSage is an algorithm for machine learning on graphs that samples the first-hop and second-hop neighbors of the vertices of the graph and then uses features of these neighbors as inputs to train a dense neural network [10]. Computationally, GraphSage is interesting because it includes both a sparse stage (multiple layers of indirection from sampling the first and second hop neighbors of a batch of vertices) and a dense stage (performing many matrix multiplications for training the dense neural network). In this evaluation, we focus on only a portion of the forward pass that isolates the sparse sampling stage and a subset of the dense stage in part as a way to demonstrate the flexibility of the Harmonic CUDA programming model without adding too much complexity to the evaluation and in part due to time constraints. We show the full forward pass and highlight our specific implementation in Figure 5.

A naive approach to GraphSage on the GPU is to simply separate the algorithm into two kernels. First, one kernel samples the first-hop and second-hop neighbors and stores their features in global memory. Next, a second kernel performs the dense linear algebra operations. This approach is suboptimal because it requires the first stage to perform an expensive global memory write of all features, followed by an expensive global memory read from the second stage to re-read all features. However, with Harmonic CUDA, it is possible to implement a more optimized approach with minimal additional complexity that combines the two kernels into a single Dataflow and stores the sampled neighbor features in intermediate on-chip buffers, only writing the final output to off-chip memory. Additionally, with Harmonic CUDA, we can reuse many of the building blocks of the previously described GEMM example in new ways to implement shared-memory linear algebra operations and can take advantage of warp specialization to run the sparse and dense stages in parallel.

5.1 Implementation

We construct a Harmonic CUDA implementation of the GraphSage algorithm using building blocks adapted from prior examples in a new context, as shown in Figure 6. The strength of Harmonic CUDA is that nowhere in this implementation do we need to explicitly specify the mapping of Nodes to compute resources or the actual implementations of any of the backends of the Nodes. By thinking of GraphSage as a dataflow graph rather than as a sequence of instructions to execute, we can easily experiment with different

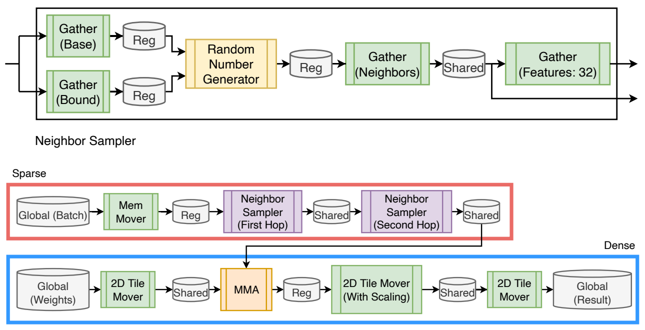

Figure 6. Top: GraphSage Neighbor Sampler Dataflow. Bottom: GraphSage Dataflow with Sparse and Dense Stages.

configurations of the Nodes and can even do more abstract reasoning about *how* to map the algorithm to the GPU. For example, by pipelining the sparse gather stage directly into the dense GEMM stage on-chip, rather than separating these into separate kernels, we can eliminate a significant amount of memory traffic. In our experiments in the following sections, we evaluate GraphSage in two configurations: one configuration where the sparse gather stage and the dense GEMM stage share the same compute resources and time slice between them, and another configuration where we assign additional threads dedicated to sparse gathers that send data to the GEMM stage over a shared memory buffer. This is easy to perform since the only changes required are to change the compute locations of the Nodes and to change the locations of the intermediate storage.

Sparse Gather Stage. In the sparse gather stage, the algorithm reads in the batch of vertices and samples 12 first-hop neighbors per batch vertex and 12 second-hop neighbors per first-hop vertex, randomly selected. A vertex with a degree less than 12 will sample duplicate neighbors. Our Harmonic CUDA implementation reuses the “MemMover” Node from Section 3 and adds several new Nodes for performing memory indirections (the “Gather” Node) and random sampling (the “RandomNumberGenerator” Node). We also create a modular “Neighbor Sampler” dataflow that we reuse multiple times for the first-hop and second-hop neighbor samples. As shown in Figure 6, we can express this algorithm as a dataflow without thinking about how to parallelize it, map it to hardware, or perform low-level optimizations. Instead, the programmer thinks about the algorithm as an abstract sequence of logical transformations done to each vertex in the batch, where data flows sequentially through the dataflow graph over time. The fact that Nodes have parallel internal implementations and produce and consume data in a pipelined manner is abstracted away from the programmer. The end result of this Dataflow is that for each vertex input to a NeighborSampler Dataflow, the Dataflow outputs a 64×32 matrix of feature values as well as a stream of the vertex indices corresponding to each matrix for each second-hop neighbor of the batch vertices.

Dense GEMM Stage. Rather than using a device-wide GEMM where all blocks work on tiles of the same matrix as seen in Section 4, our implementation of GraphSage uses a per-block GEMM where each block repeatedly multiplies and accumulates samples of the second-hop vertex features with a small weight matrix. Additionally, where in our GEMM implementation in Section 4 we previously mapped the MMA Node to the entire block, in GraphSage, we can instead map the MMA Node to only a portion of the block, leaving the rest of the block free to perform the neighbor indirections to collect the features. We also have different 2D tile movement patterns to consider. Where we previously iterated over tiles of B in global and shared memory, GraphSage instead repeatedly uses the weight matrix as B, with the A matrix changing each iteration. Harmonic CUDA's abstractions allow us to easily reuse the existing building blocks, where the only necessary change is a small modification to the flags that describe where the MMA Node runs, the location of the inputs, and the size of the inputs.

5.2 Results

We compare a warp-specialized, asynchronously-scheduled GraphSage implementation in Harmonic CUDA to a bulk-synchronous, interleaved configuration in Harmonic CUDA, showing significant performance improvements and easier experimentation. To prove these speedups are from more efficient hardware utilization, rather than additional hardware resources allocated during warp specialization, we also compare against cuBLAS using a variant of GraphSage that eliminates sparse lookups to isolate the performance of the dense GEMM stage.

GraphSage Performance. Our first Harmonic CUDA implementation is bulk-synchronous and interleaved. In it, we assign all Nodes in the Dataflow to a full block (256 threads) and interleave sparse lookups with dense GEMM operations, yielding a throughput of 8.78 TFLOPS. Harmonic CUDA's flexibility allows us to easily pivot our implementation to a different configuration. In this implementation, we assign sparse lookups to an additional 64 threads, while keeping the GEMM on 256 separate threads. This allows asynchronous scheduling with data passing over a double-buffered shared memory Connector and improves throughput by 34% to 11.75 TFLOPS. The only thing we have to do to make this change is to modify the single parameter that specifies which Cooperative Group each Node runs on.

Now, does this performance improvement stem from the warp specialization approach or from the additional 64 threads per block? Harmonic CUDA's flexibility aids us in performing this design exploration. To begin, we modify GraphSage to eliminate indirection from first- and second-hop neighbor lookups, instead repeatedly using only batch vertex features.

This yields a throughput of 13.73 TFLOPS. Each block now only performs a series of small GEMM operations to repeatedly multiply the batch feature matrices with the weight matrix. If this GEMM-like kernel can perform on par with cuBLAS, we can conclude that because cuBLAS is able to efficiently saturate the hardware, adding additional threads to this modified GraphSage experiment would not improve performance.

GEMM Performance. We approximate the repeated small GEMM operations of the modified GraphSage kernel as if they were tiles of a larger GEMM. Testing the equivalent matrix[1] in cuBLAS, we achieve 14.6 TFLOPS. Since this is only a 6% improvement over the modified GraphSage kernel, we can conclude that the GraphSage GEMM stage would not benefit from allocating an additional 64 threads (a 25% increase) to the stage.

Analysis. This GraphSage example highlights the advantages of considering the algorithm's dataflow and computation mapping separately. With Harmonic CUDA, the programmer can easily experiment with different mappings of Nodes to compute resources. By relocating each Node and the intermediate buffer storage with a simple parameter change, the formerly bulk-synchronous kernel becomes warp-specialized, resulting in improved performance compared to the bulk-synchronous version. Additionally, by viewing GraphSage as a Dataflow of building blocks, we can easily express changes in computation and storage locations, keep intermediate results on-chip, and reuse the building blocks from Section 4 in a new context.

6 Conclusion

Harmonic CUDA demonstrates that rather than thinking about GPU code as a sequence of operations to be executed by a thread, the programmer can reason about how an algorithm transforms data without needing to worry about how specifically to map the algorithm to the GPU. One of Harmonic CUDA's most important ideas is that any building block has use cases in many contexts that are not traditionally supported by GPU libraries and that a programmer should be able to reuse the same building block in many applications and at many different granularities. We bring this programming model to bear on two real-world examples: Matrix Multiplication and GraphSage.

We identify several areas for future research. First, expanding Harmonic CUDA to new algorithms that focus on irregular parallelism, specializations of subsets of a grid, and algorithms that require more complex Dataflow graphs. We also believe that Harmonic CUDA should expand beyond a single GPU to groups of multiple GPUs on one Node or other architectures such as CPUs and accelerators. This would allow the *where* abstraction to extend not just to layers of the GPU's memory hierarchy, but to CPU cores, domain-specific hardware, and heterogeneous systems. It should also expand to clusters of multiple GPUs or multiple Nodes, where it has the potential to make managing data orchestration in a distributed system easier. In addition, we would like to further improve the model to make it as performant and easy-to-use as possible.

[1]The equivalent matrix has M as the total batch vertices (108 blocks $\times$ batch size of 64), N as the weights per feature (64), and K as the total number of features (12 first-hop neighbors $\times$ 12 second-hop neighbors per first-hop neighbor $\times$ 32 features per second-hop neighbor) for $(M, N, K) = (6192, 64, 4608)$.

Acknowledgments

This work was largely done at NVIDIA under the guidance of Steve Keckler. Thank you also to Angshuman Parashar for Figure 5 and to Cris Cecka for help with profiling the GEMM kernels. This research was, in part, funded by the U.S. Government under the DARPA SDH program. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] Farhoosh Alghabi, Ulrich Schipper, and Andreas Kolb. 2014. A Scalable Software Framework for Stateful Stream Data Processing on Multiple GPUs and Applications. In *GPU Computing and Applications*. Springer Singapore, 99–118. https://doi.org/10.1007/978-981-287-134-3_7

[2] Michael Bauer, Henry Cook, and Brucek Khailany. 2011. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. 1–11. https://doi.org/10.1145/2063384.2063400

[3] Michael Bauer, Sean Treichler, and Alex Aiken. 2014. 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)*. 119–130. https://doi.org/10.1145/2555243.2555258

[4] Jack Choquette, Oliver Giroux, and Denis Foley. 2018. Volta: Performance and Programmability. *IEEE Micro* 38, 2 (April 2018), 42–52. https://doi.org/10.1109/MM.2018.022071134

[5] Federico Ciccozzi, Lorenzo Addazi, Sara Abbaspour Asadollah, Björn Lisper, Abu Naser Masud, and Saad Mubeen. 2022. A Comprehensive Exploration of Languages for Parallel Computing. *ACM Comput. Surv.* 55, 2, Article 24 (Jan. 2022), 39 pages. https://doi.org/10.1145/3485008

[6] William J. Dally, Stephen W. Keckler, and David B. Kirk. 2021. Evolution of the Graphics Processing Unit (GPU). *IEEE Micro* 41, 6 (2021), 42–51. https://doi.org/10.1109/MM.2021.3113475

[7] Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, and Bryce Adelstein Lelbach. 2022. std::execution. C++ Standards Committee Papers. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2300r5.html

[8] Alex Fender, Brad Rees, and Joe Eaton. 2022. RAPIDS cuGraph. In *Massive Graph Analytics*. Chapman and Hall/CRC, 483–493. https://doi.org/10.1201/9781003033707-22

[9] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture.* https://doi.

org/10.1109/micro.2016.7783759

[10] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. 1025–1035. https://proceedings.neurips.cc/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf

[11] Mark Harris and Kyrylo Perelygin. 2017. Cooperative Groups: Flexible CUDA Thread Programming. https://developer.nvidia.com/blog/cooperative-groups/

[12] Kartik Hegde, Hadi Asghari Moghaddam, Michael Pellauer, Neal Clayton Crago, Aamer Jaleel, Edgar Solomonik, Joel S. Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*. 319–333. https://doi.org/10.1145/3352460.3358275

[13] Dominique Houzet, Sylvain Huet, and Anis Rahman. 2010. SysCellC: a data-flow programming model on multi-GPU. *Procedia Computer Science* 1, 1 (May 2010), 1035–1044. https://doi.org/10.1016/j.procs.2010.04.115 ICCS 2010.

[14] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. 2017. CUTLASS: Fast Linear Algebra in CUDA C++. https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/

[15] Ronny Krashinsky, Olivier Giroux, Stephen Jones, Nick Stam, and Sridhar Ramaswamy. 2020. NVIDIA Ampere Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/.

[16] MathWorks Corporation. 2022. Simulink. https://www.mathworks.com/help/simulink/index.html

[17] Duane Merrill. 2013–2022. CUB: Flexible Library of Cooperative Threadblock Primitives and Other Utilities for CUDA Kernel Programming. (2013–2022). https://github.com/NVIDIA/cub.

[18] National Instruments Corporation. 2022. LabVIEW Documentation. https://www.ni.com/docs/en-US/bundle/labview/page/lvhelp/labview_help.html

[19] NVIDIA Corporation. 2020. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-tensor-core.

[20] NVIDIA Corporation. 2022. CUDA cuBLAS Library (v11.6). http://developer.nvidia.com/cublas.

[21] NVIDIA Corporation. 2022. CUDA Samples. https://github.com/NVIDIA/cuda-samples.

[22] NVIDIA Corporation. 2022. libcu++: The C++ Standard Library for Your Entire System. https://nvidia.github.io/libcudacxx/ Version 1.8.1.

[23] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer. 2019. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 137–151. https://doi.org/10.1145/3297858.3304025

[24] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM Press, 519–530. https://doi.org/10.1145/2491956.2462176

[25] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, R. Nigel Horspool (Ed.). Springer-Verlag, 179–196. https://doi.org/10.1007/3-540-45937-5_14