Optimizing scheduling for stream structured programming for StreamIt

by

Nicholas Lee Dow

S.B. in Computer Science and Engineering, MIT, 2024

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

Authored by: Nicholas Lee Dow
Computer Science and Engineering
May 16, 2025

Certified by: Saman P. Amarasinghe
Professor of Electrical Engineering and Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurtis
Chair, Master of Engineering Thesis Committee

Optimizing scheduling for stream structured programming for StreamIt

by

Nicholas Lee Dow

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

As straightforward increases in performance on general purpose CPUs slow down, the shift to application specific implementations and hardware has accelerated. This shift to towards specialization improves performance but often at the cost of developer productivity in learning these new tools. StreamIt is a Domain Specific Language developed to increase performance of streaming applications while being relatively user-friendly. While designed to be parallelized easily, the scheduling backend of the StreamIt compiler is not adapted to the heterogeneous and distributed nature of new accelerator hardware. This thesis details the design and development of a scheduler interface that enables hardware customized schedulers to be developed quickly. The scheduler interface allows for schedulers to take advantage of the unique compiler optimizations enabled by StreamIt's structure. Two schedulers, one search based and another heuristic based, are built using this interface to schedule StreamIt workloads to optimize differing metrics such as throughput and latency. Our experiments evaluate the performance of these workloads, and details future direction for expanding the interface and scheduler designs that could take advantage of it.

Thesis supervisor: Saman P. Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I'd like to thank Professor Saman Amarasinghe and Ajay Brahmakshatriya for guiding me through the intricacies of stream programming with wisdom and patience, and for starting my journey in performance engineering as inspiring teaching staff for 6.106(then 6.172). I'd also like to thank Tasmeem Reza for being my fellow StreamIt compiler expert.
Thank you to my parents for their unconditional support, my siblings for their competitive spirit, and my friends for putting wind in my sails. You've made my trip to and through MIT the best it could've been.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the slowing down of Moore's law, improvements in speedups of high-performance computations has slowed down precipitously. Even with the recent rush into machine learning computation with the rise in popularity of large language models, each successive chip generation provides less improvement over the last. When gains are made in hardware, they are also no longer purely a clock-speed or core count increase, but now more often rely on specialized techniques to achieve their gains. Advancements such as asynchronous memory streaming and lower bit-count arithematic precision require complex adaptations by software developers to take advantage of these new gains.

Thus, the era of simply porting over old code to new hardware and expecting improvements is over, requiring software performance engineering to take up more importance in engineering a well-performing application. While coding by hand is still necessary to bridge the gap to state of the art performance, industry and academic groups are increasingly relying on Domain Specific Languages (DSLs) to provide state of the art performance out of the box. DSLs are designed with specific application domains in mind, and allow compiler experts to encode domain expertise into the interface and code generation of the DSL to produce code on par to hand-written performance engineered code.

Popular and powerful examples of this trend include the GPU scheduling language Triton,

which provides block structured abstraction to ML scientists to build models that can be compiled from it's pythonic representation to efficient CUDA code[1]. Another such example is Halide, a DSL targeting images and tensors that exposes a scheduling and loop nest interface to allow for developers to quickly iterate over scheduling possibilities for the specific hardware they are targeting, allowing the same algorithms to be quickly ported to other platforms[2].

One such DSL developed at MIT and the main subject of this thesis is the StreamIt language [3][4]. StreamIt was developed specifically to increase the performance of programs that rely on 'stream programming', defined as programs that continuously process input data to perform calculations indefinitely. It leverages static declarations of data-flow and discretization of computation into a composed graph of filters to perform aggressive compiler optimizations to automatically parallelize stream programs.

The structure of StreamIt programs allows easy parallelization of programs, but raises the problem of *scheduling* the program. In the context of StreamIt, *scheduling* is defined as finding the optimal mapping of sub-processes of a StreamIt program to hardware resources. StreamIt's compiler parallelization optimizations change the number and compute intensity of these sub-processes, so resource allocation and optimal compiler transformations to StreamIt program are intertwined.

The original StreamIt work and it's associated scheduler was based on the RAW architecture, a heavily parallel hardware set-up where identical processing cores are built in a grid with communication to only their immediate neighbors.[4] While RAW is great for concretely analyzing parallelization methods and communication constraints, the scheduling methods developed for RAW do not readily generalize to the swath of hardware platforms that stream processing is being developed on; RAW was a homogenous architecture made of tiled, indentical cores while today's state of the art hardware is largely heterogenous. Modern architectures for stream programming usually include interconnected CPUs and GPUs, with larger setups requiring distributed processing. The advent of these new hardware paradigms

introduces the need for a way to develop and prototype new scheduling algorithms that can adapt StreamIt applications to different hardware platforms.

1.1 Problem Statement

The goal of this work is to design and develop an interface that scheduling algorithms can use to schedule StreamIt applications on a variety of hardware platforms that optimize scheduler defined metrics. This interface will be tested by schedulers implemented as a part of this project, with throughput and latency being the metrics evaluated. The schedulers will be optimizing these metrics by load-balancing hardware resources among the computation costs of StreamIt graphs. The interface is designed for the future additions of StreamIt specific compiler optimizations that scheduling influences, with a couple of these optimizations implemented for the schedulers to leverage.

1.1.1 Motivating Example

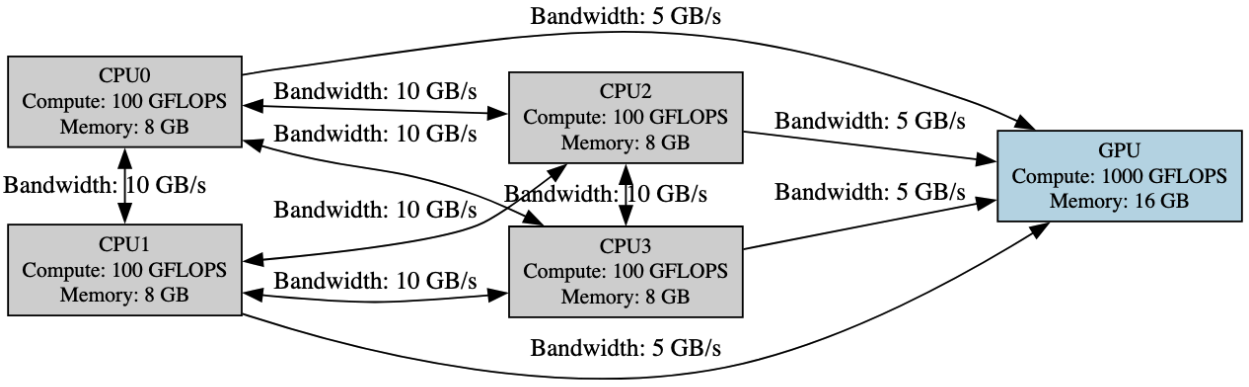


Figure 1.1: Example of hardware graph abstraction.

Here we depict a motivating example of the scheduling problem. In figure 1.1 there is a set of CPU cores in a fully connected mesh pattern, with each having an independent bandwidth connection to a GPU. There are descriptions of various resources like compute, bandwidth,

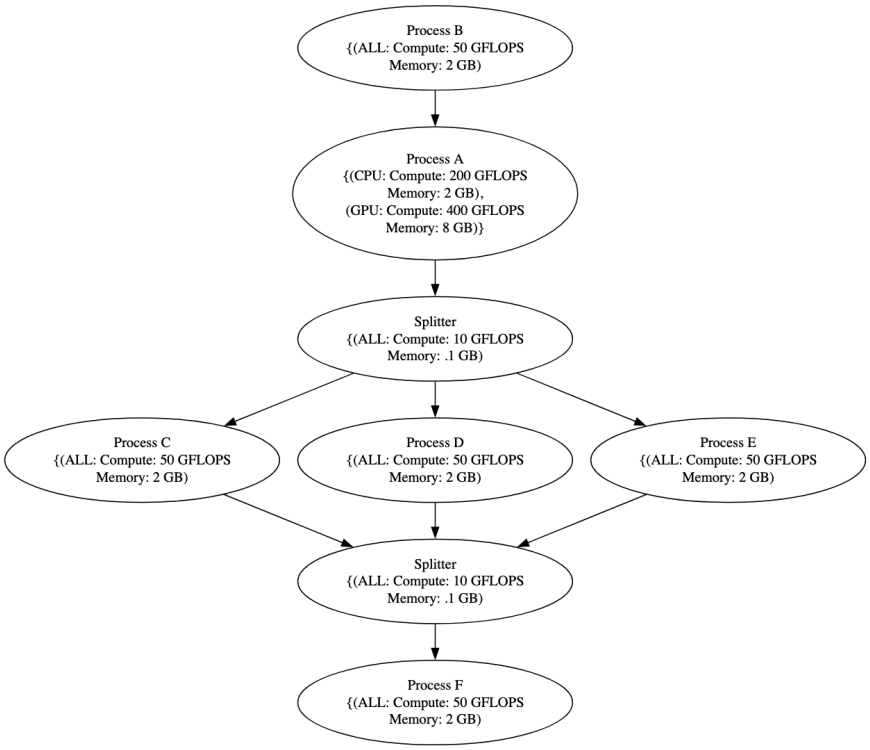


Figure 1.2: Example of computation DAG with costs.

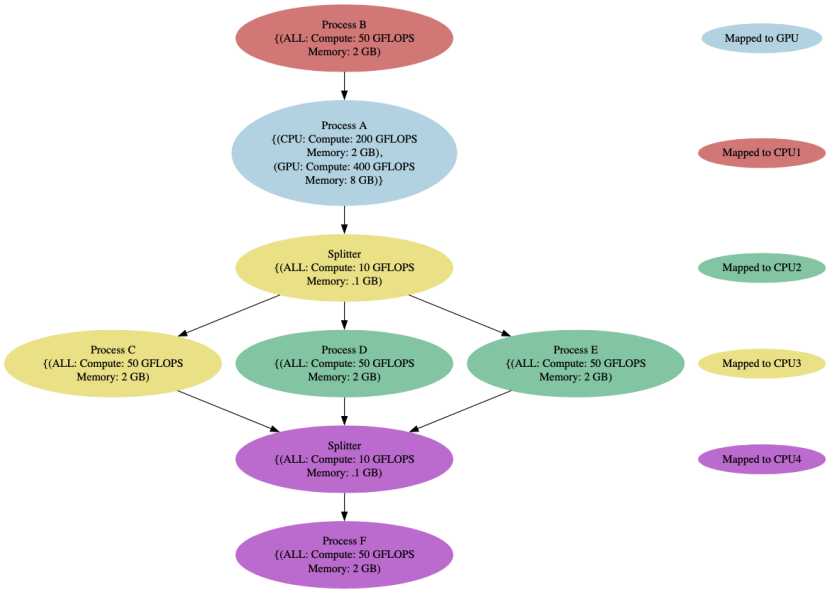


Figure 1.3: Mapped computation DAG to hardware graph.

and memory, with the GPU being substantially more powerful but with lower bandwidth to the CPUs. This is the target architecture we wish to map the StreamIt computation DAG of 1.2 in a manner that maximizes a certain metric, in this case throughput. Throughput is maximized if the nodes of figure 1.2 are mapped in a load-balanced manner, as throughput is limited by the slowest hardware unit to complete it's respective computation.

Figure 1.3 represents a load-balanced mapping of the computation to hardware, labeled by color. Intuitively, this mapping was generated by a greedy algorithm that mapped the most compute intensive nodes first to the hardware that experienced the lowest increase in their load. The load increase can vary between units due to both resources available and that nodes can have different costs to run on different hardware. This is shown in the mapping of Process A to the GPU, as the compute cost of running A is double of running it on the GPU, but the higher clock of the GPU means that the load is only 40% rather than 200% on a CPU.

As nodes are mapped to hardware, the scheduler should prioritize mapping nodes to hardware where their neighbors in the computation graph have already been mapped, as this minimizes communication costs and allows for computation optimizations to be made. Effective schedulers and a well-designed interface for schedulers to be built upon is essential for increasing the performance of StreamIt programs on heterogenous hardware.

1.2 Overview of Thesis

Chapter 2 of this thesis discusses some related work to the topic of scheduling oriented DSLs and scheduling in StreamIt. Chapter 3 details the basics of the StreamIt language needed to understand the constraints of scheduling StreamIt programs. Chapter 4 delves into the design and implementation of the scheduling interface and two schedulers built upon the interface. In Chapter 5 we evaluate the performance of the scheduling algorithms in fitting to constraints and real world performance of the scheduled output. Chapter 6 details

additional optimizations and features implemented for StreamIt as a part of this thesis. All implementations and materials relating to the content of this thesis can be found in the StreamIt repository[5].

Chapter 2

Related Works

In this section we will examine some of the prior work of scheduling StreamIt. We will also look at other DSLs that expose explicit scheduling decisions to developers to target a swath of hardware.

2.1 Prior StreamIt Work

As discussed in Chapter 1, scheduling for StreamIt initially targeted the RAW hardware architecture. Because the architecture was a mesh of identical cores, the strategy they employed was to create the same number of partitions of StreamIt nodes as cores, and to have the compute cost of each to be roughly equal. Given changing hardware targets, this strategy is not effective at achieving peak performance outside those restrictions. Additionally, they required that all nodes mapped to the same hardware to be connected in the computation graph. With some code being specialized for some hardware and sometimes even unable to be run on some of the hardware nodes of the system, relaxing this requirement would allow for some StreamIt programs to be scheduled that would not have been supported otherwise.[3][4]

Some work has been performed in online, dynamic scheduling of streamit nodes to processors using an economic agent model to drive assignment [6]. Dynamic scheduling allows the execution to response to computational load while static scheduling cannot, but restricts

the scheduler to allocation decisions only instead of allowing transformations to the actual graph.

2.2 Halide

Halide is a domain-specific language (DSL) designed for high-performance image and array processing, which separates algorithm specification from execution scheduling. Its scheduling model allows users to explicitly control how computations are mapped to hardware through a set of scheduling primitives such as tiling, vectorization, parallelization, and loop reordering. This decoupling enables a clear distinction between the functional description of a computation and the strategy used to optimize its performance.[2] Unlike compilers that rely on automatic scheduling heuristics, Halide empowers developers to manually explore and fine-tune different scheduling strategies, often achieving performance competitive with hand-optimized code. Subsequent work has extended Halide with autoscheduling capabilities that leverage cost models to automate this process, improving usability while preserving performance benefits. [7] Our work reflects a similar design philosophy of separating the computation from it's optimal mapping to hardware, but with constraints specific to the streaming domain.

Chapter 3

The StreamIt Language

3.1 The StreamIt Language

The StreamIt programming language is a DSL for stream programming that aims to increase performance of applications by encoding streaming abstractions into the structure of the programming language. [3][4]. By providing a higher level descriptions of the work of algorithm and implicitly providing the data-flow of algorithms, the compiler is able to make more aggressive and performant optimizations that is out of reach of traditional compilers. By focusing on 'stream programming', StreamIt is especially equipped for optimizing applications within signal processing and machine learning.

3.1.1 Language Overview

The universal component in StreamIt is the stream; the generic description of a stream is an object that in the course of a single execution takes in data, performs some computation, and outputs data. In StreamIt, there are three primary types of stream constructs that help define and organize data flow within streaming applications: filters, pipelines, and split-joins. Each of these provides a distinct way to structure computation and communication within a streaming algorithm, and can be composed to express complex applications.

The most basic type of the streams, and the one that primarily performs computation, is the *filter*. Filters have a single input channel, a single output channel, and associated functions that describe initialization of the filter and the continuous computation the filter performs when it runs: *init* and *work* functions respectively. Common operations that *init* might perform are setting up state that the filter might need, such as internal buffers of parameters. The channels for all streams are first-in first-out (FIFO) queues.

The *work* functions must declare *pop*, *peek* and *push* rates. These are parameters that expose the data consumption pattern when the filter *fires*, the term for a single atomic execution of the filter's work function. The *pop* rate is the number of data items that the *work* function consumes in a single execution, and correspondingly is the number of items it "pops" from it's input channel using the `pop()` function. The *peek* rate is depth the of data items that the work function reads in the input channel to perform a computation using the function `peek(index)`, where `index` is the position of the input channel `index` items from the next value to be read. Therefore the peek rate of any filter is greater than or equal to it's pop rate, with *peek* rates being useful for filters that produce rolling averages, for example. Peek rates also allow for filters that would normally have state to be able to externalize the state into peeking buffers, allowing for more aggressive compiler optimizations. The *push* rate is the number of data items that the work function "pushes" onto it's output channel each iteration.

Split-joins and pipelines are both streams that allow the composing of streams to build full StreamIt applications and do not perform computation beyond data movement. *Pipelines* simply allow serial composition of streams such that the outputs of the nth stream of the pipeline are the inputs of stream $n + 1$. *Splitjoins* allows the splitting of data by creating a *splitter* that splits the input data among the composed streams it contains. The outputs of the composed streams are combined by a *joiner* for single output channel of the splitjoin. *Duplicate* splitjoin send every data item to every stream, essentially acting as a broadcast function of the data. *Roundrobin* splitjoin send each data item to a single stream by starting

with the first stream and sending a specified amount of data, then the second stream, and so on until wrapping to the first stream again. The amount sent to each is specified by a set of parameters, which default to 1. *Joiners* of both types of splitjoin are the same and behave similarly to the roundrobin *splitter*, where the joiner pulls a specified amount of data from each output channel each round, defaulting to 1.

The StreamIt code to construct the motivating example shown in figure 1.2 is shown in the code snippet 6.1. The example is a pipeline of mainly filters, with a single duplicate split-join that has 3 child streams, all of which are filters.

```
auto pipe = pipeline<int, int>([&] (auto p) {
    auto source = input<int>(1);
    auto b = process<int, int>();
    auto a = big_process<int, int>();
    auto sj = duplicateSplitJoin<int, int> ([&] (auto sj) {
        auto c = process<int, int>();
        auto d = process<int, int>();
        auto e = process<int, int>();
    });
    auto f = process<int, int>();
    auto sink = pure<int, int>(1, 0, [&] (auto in, auto out) {
        writeComplex(in->pop());
    });
    p->add(source);
    p->add(b);
    p->add(a);
    p->add(sj);
    p->add(f);
    p->add(sink);
});
```

Listing 3.1: StreamIt code to recreate the motivating example figure.

3.1.2 Steady State Schedule

A StreamIt schedule is a ordered list firing of the nodes in a StreamIt graph.[8] During a schedule, each StreamIt node consumes data from it's input buffer and pushes data onto it's output buffer. An essential concept to general signal processing and how StreamIt works is the steady state schedule. A steady state schedule is a schedule of computation that can continue to execute indefinitely. An important property of the steady state schedule is that

the data occupancy of buffers between all filters are the same before and after the execution of the schedule; otherwise buffers would either reach capacity or not have enough data after successive executions of the schedule, thus violating our requirement for it to be able to execute indefinitely.

A steady state is defined as the number of times each stream needs to execute in a steady state schedule[8]. Due to non-matching push and pop rates within splitjoins, it's possible for a StreamIt program to not have a steady state schedule; these programs are not valid StreamIt programs. Steady state schedules are calculated first by generating a Minimal Steady State as defined in [8], and then running a pull schedule on the StreamIt graph using the number of firings of each node in the minimal steady state to ensure all nodes are fired sufficiently. While there are multiple different ways to generate steady state schedules, this method prioritizes smaller buffers by ordering filter firings such that they fire as soon as they have sufficient data in their input buffer through the pull schedule. The minimal steady state is essential for partitioning the nodes to different hardware units, as the it shows the frequency of filter firings and data movement in steady state, which combined with compute costs gives an accurate estimate of actual compute utilization of each StreamIt node in steady state.

3.1.3 Compiler Optimizations

StreamIt enforces a structure where the programmer defines their algorithm as hierarchy of composed basic filters. By breaking up the computation and explicitly defining data flow, programmer enables the compiler to perform a variety of different optimizations to fit the programmer's hardware target. After given a program and hardware specification, it is up to the compiler's scheduler to adapt the program to the hardware by parallelizing and partitioning the nodes of the StreamIt graph to available hardware units. Here, we define partitioning to mean the load-balancing the allocation of nodes to hardware resources such that the schedule optimizes a given metric, such as throughput.

The most straightforward of these compiler optimizations is fusion. Fusion is when a group of contiguous StreamIt filters are grouped together during scheduling, allowing for cache and synchronization optimizations as they will be compiled as a single executable during code generation. One such optimization implemented in the compiler is the use of circular buffers rather than traditional queues between fused nodes[9].

Fission, traditionally called vertical fission in previous StreamIt papers, is the splitting of a particularly compute intensive filter into a composition of filters that performed the same function. This is quite hard to do with static analysis, so it is a feature where programmers would have to define how a fission would work. There is one such variant of the stream that is currently in development that has this feature implicitly, called the parameterized splitjoin. It has a user defined splitting function and parameterized child streams that increase or decrease the number of child streams as a parameter n is changed, and is a way for the compiler to perform such a fission.

Called horizontal fission in several of the original StreamIt papers, we call this compiler optimization Scaling Up. Due to the emphasis of peeking and uses of data references in StreamIt, many filters can be duplicated across hardware units to provide parallelization with little static analysis. This transformation enables higher throughput in parts of the StreamIt graph that might be bottle necked by a compute intensive node.

3.1.4 Portal Messaging

Portal Messaging is a feature in StreamIt to allow infrequent, data-flow independent control messages to be passed between nodes of the StreamIt computation graph. When they were developing StreamIt, the original authors of the language noticed a significant number of applications required control messages to be passed between nodes. The initial solution was to include extra fields in the data being passed between nodes, but since the messages were infrequent and dynamic the control data fields were mostly unused during execution. Portal Messaging was introduced to solve this problem with the additional benefit of being able to

send messages to prior nodes in the computation graph[3].

Portal Messaging functions similar to regular data-flow in that it has a FIFO queue between the sender and the receiver, but the buffer does not have static push and pop rates. Instead, it has a parameter called `latency` that essentially dictates when a node *acts* upon a message. Latency allows the user to specify destination nodes for portal messaging topologically above and below the source node in the computation graph, with the restriction that the message can not cause the destination node to act upon data that could have triggered the sending of the *same message*. In brief, portal messaging is restricted to not cause cycles in data-flow. The fine details of latency are not needed to understand this thesis, but the important point of latency is that it allows the compiler to check for valid data-flow of messages.

In the figure 3.1, we present an example of portal messaging between many detector nodes to node RFtoIF, upstream of them in terms of data-flow. A latency of 6 basically means that the message will act on data that is 6 firings of the source node after the data used to send the message, prevents a cycle of data.

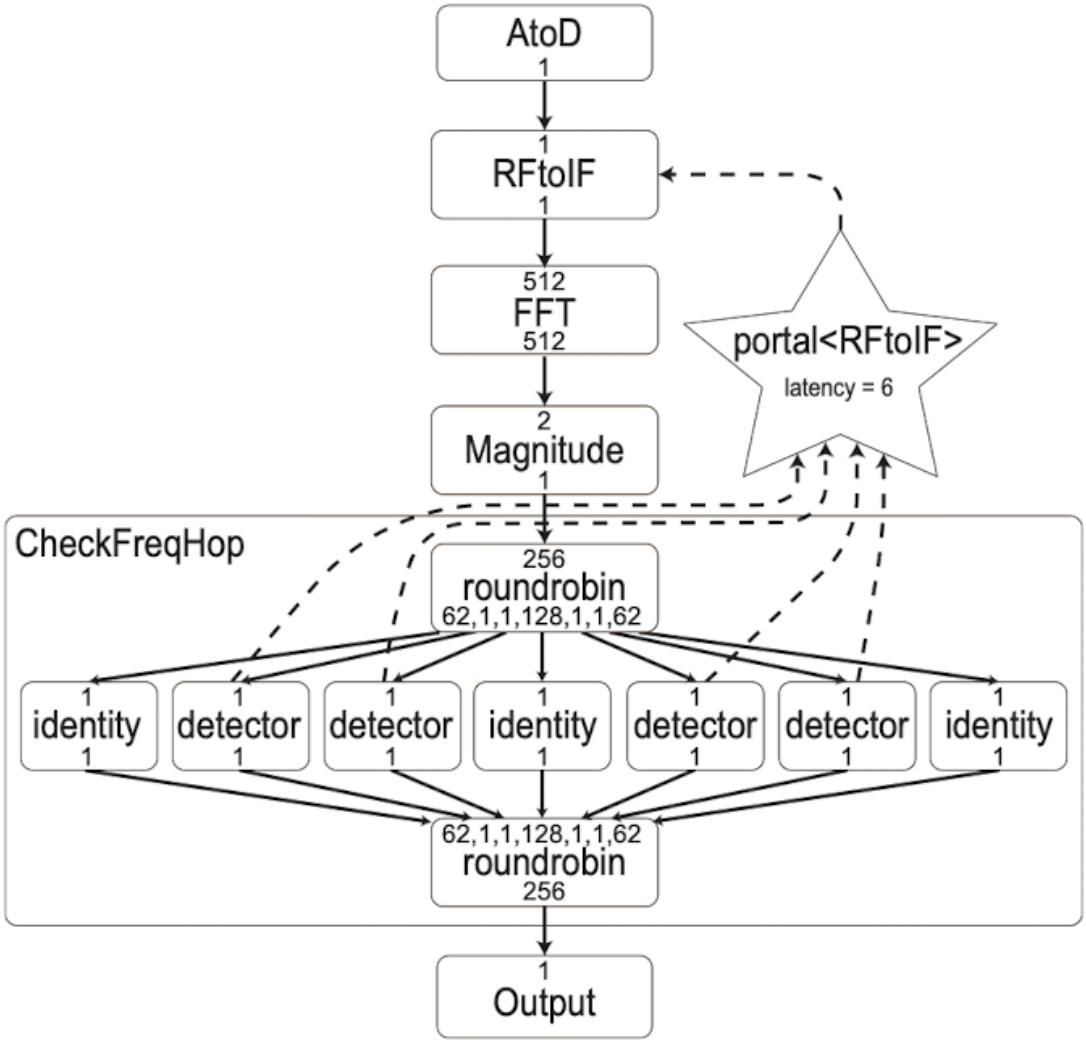


Figure 3.1: Depiction of Radio-hopping application with portal messaging from [3]

Chapter 4

Static Scheduling Interface for StreamIt

In this chapter we detail the design motivations for the scheduling interface, driven by adaptation to new hardware. I discuss the general role of the interface in scheduling and enumerate the API endpoints of the interface. I discuss a key cycle-breaking algorithm I developed in enabling the flexibility of schedulers, and detail the implementation of two schedulers.

4.1 Introduction

The StreamIt language's design, as discussed in Chapter 3 and other works, allows parallelization opportunities not explicitly outlined by the user. Sections of the StreamIt computation graphs DAG structure identified to have long latency of computation can be parallelized across available hardware units.

However, these scheduling decisions are not always clearly optimal, and often require analysis of both the properties of the code of the computation graph as well as the properties of the hardware that the StreamIt graph is being compiled to be ran on. a A scheduling decision on whether to copy a computationally intensive node across multiple cores of a single processor would be complicated by that same node having an a high throughput stream of output data that would slow down the entire computation despite parallelization if the next

node we not co-located on the same core. Some hardware also has asymmetric communication patterns, where the parallelization example above would improve performance if the copy of the node was located on another compute unit with a low latency to the core with the next node, but would suffer on a different unit.

This scheduling problem would be complex enough if the target hardware were relatively uniform in terms of computational capability, e.g. TeraFLOPs of compute capacity, and network topology with uniform networking between compute units; examples of this would be the structure of a multi-core CPU or a GPU compute cluster. However, the computations that StreamIt targets often span heterogenous hardware, where general purpose CPUs are often combined with accelerators for specific tasks. One such example is the Tracer project, in which a fork of StreamIt is being developed for FPGAs with both FFT accelerators and general purpose processing elements.

To solve this problem, we sought a way to build efficient scheduling algorithms that could not only bridge the gap between the varying constraints of different hardware architectures and allocating Streamit graphs effectively, but also leverage StreamIt's static graph structure to transform the computation graph to suit the target hardware. We also acknowledged that different scheduling algorithms would likely need to be developed for different hardware architectures to fully leverage their capability. Not all applications call for optimizing the same metric, so generalizing this interface also allows for schedulers to be designed for different objectives in mind. A scheduling algorithm for running PyTorch kernels on GPUs might wish to optimize throughput, while a signal processing algorithm might wish to optimize the average latency of the computation above all else.

4.1.1 The Problem

This led us to develop a scheduling algorithm interface that manages the resources of hardware, calculates computational costs of StreamIt nodes, and effects StreamIt graph transformations such that scheduling algorithms can built on top. We then use this interface to build a

couple scheduling algorithms that leverage StreamIt's graph abstraction to produce a correct mapping of streams to hardware.

4.2 Hardware graph and Cost Model

Graphs are useful tools for modeling network topologies and are used in literature to model hardware in a similar sense to how which we wish to model the problem, so we choose a graph abstraction to model hardware resource constraints and to map elements of the StreamIt graph to hardware. We represent units that perform any sort of computation as *compute nodes* in our `HardwareGraph`. To support the range of possible processor units, compute nodes can have arbitrary types as shown in earlier figure 1.1.

The edges of the `HardwareGraph` represent the *data-flow* between the compute nodes of the graph. These can also be arbitrarily typed to be able to represent the variety of hardware. The connections these data-flow edges model can range from shared L3 cache on an Intel Xeon CPU to the DRAM link between a host CPU and a GPU.

These arbitrary types also allow us to specify different resources costs that a compute node might consume on different hardware. We introduce the `ConstraintMap` that's associated with each stream in a StreamIt graph, and contains the resources that node would consume if scheduled on a compute node type. These constraints can be distinct for each type of hardware type, and can also be used to specify that a given node is unschedulable on a specific hardware type. These constraint types can also be specified to be able to support limited allocation of nodes that consume a specific kinds of resources, such as tensor cores on a GPU.

Constraint Types. There are two supported generic constraint categories in the scheduler interface: *temporal* and *static*. These constraints are used for specifying both compute node resources and data edge resources. Temporal constraints represent any resource that is time-bounded and which streams continuously consume to run, with the primary example

being FLOPs for compute nodes and bandwidth for data-flow edges. Static resources represent resources that do not change overtime and whose consumption does not change with how often streams run; the primary example would be memory. To fully specify the constraints that a unit of memory might impose on a computation, it might be necessary to specify both a compute node's total memory storage and it's read and write bandwidth, shown in a more specified version of our initial hardware example in figure 4.1. Also shown are capabilities unique to some hardware, such as the GPU's capacity for asynchronous reads from CPU host memory allowing a developer to specify that a version of a filter can utilize that feature in the filter's `ConstraintSet`[10].

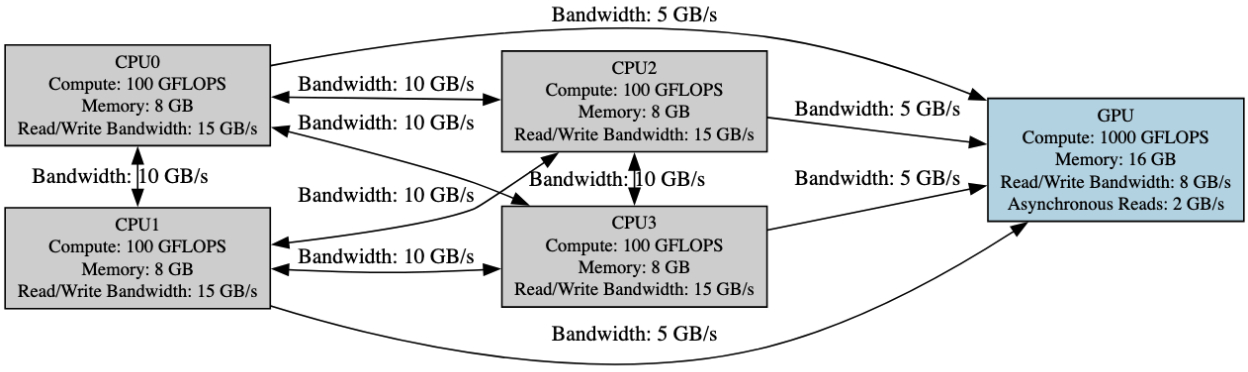


Figure 4.1: Example of hardware graph abstraction with more specification of constraints and unique resources.

The resources used by all the StreamIt nodes allocated to a hardware unit A are determined by summing up the `ConstraintSet` associated with $A's$ hardware type. While hardware utilization does not always occur in this additive fashion, as problems such as cache thrashing can cause runtime to slow more than the sum of the two node's runtimes, we deemed that this simplification was adequate to roughly model the constraints posed by hardware. The same logic applies to how we calculate data-flow edge constraints, except that the assignment of StreamIt edges to data-flow edges is implicitly determined by the allocation of the StreamIt nodes they connect. StreamIt nodes on the same hardware unit would force the edge to use memory if enough data needs to be buffered, and nodes on separate hardware units would

force the StreamIt edge to be allocated to the data-flow edge between the units.

We believe that these two generic categories can specify nearly all constraints necessary for the scheduler to provide a stream-to-hardware mapping that both correct in that no resource is over-allocated and can optimize a number of important performance metrics well.

4.2.1 Scheduling Problem

The scheduler interface is designed to allow design freedom for schedulers while encoding constraints necessary for correct transformations and hardware mappings. In terms of steps to code generation, the scheduler sits between a developer built StreamIt *virtual graph* and code generation for hardware targets. We define a StreamIt *virtual graph* as a graph that contains all the information of the StreamIt algorithm to be compiled and the hardware resources each node needs, but is still able to be transformed with the aforementioned StreamIt compiler optimizations. A StreamIt scheduler uses the scheduler interface to perform transformations on the graph and assign nodes to hardware units, and after scheduling, the scheduling interface outputs a stream to hardware mapping and a *concrete graph* that has been transformed by the scheduler for this hardware graph.

4.2.2 How The Interface Works

The scheduler interface was built with both usability and flexibility in mind for a developer to make a new scheduler to suit their needs. A custom scheduler might be needed for a specific hardware architecture, or an application with performance behavior that is sufficiently unique to warrant a different algorithm than one provided. The overarching philosophy of the scheduler is to abstract most of the cost calculations, evaluation of constraints, and graph transformations away from the developer. While the scheduling algorithm inherently needs to have access to the cost of assigning nodes to specific hardware to evaluate choices of hardware, it's the scheduling interface's job to maintain the current assignment of streams to hardware and the current resource load on each hardware unit. This abstraction allows the

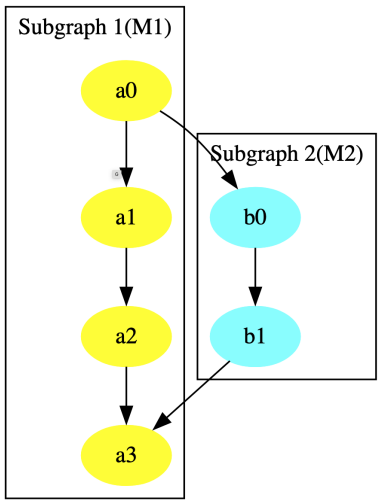


Figure 4.2: An example of a cyclic dependency.

scheduler interface to internally verify that the stream to hardware mapping is always valid and constrain some stream to hardware assignments that would not be valid with respect to StreamIt's structure without forcing a developer to include that logic in their scheduling algorithm.

The first important consideration that is abstracted away by the scheduling interface is the impact of the steady state schedule on the StreamIt nodes actual cost during runtime. While each node and edge have costs associated with single firings, the number of firings a node might have during each steady state schedule will multiply this cost. By calculating the minimum steady state as discussed in 3.1.2, the scheduler interface can present the true cost of a node to a scheduling algorithm. The same logic applies to managing data-flow edge constraints, where the interface can statically calculate the steady state usage of each edge to determine the true cost of placing two connected nodes on different hardware. An example of where this might come into play might be an instance where an assignment introduces a cyclic dependency between two sets of streams mapped to different hardware. For this example, let's call these two mappings M_1 and M_2 and in this instance they are connected subgraphs of nodes in the original StreamIt graph. If an output of a node assigned to M_1 is upstream of a input in M_2, and then an output of M_2 is upstream of a input in M_1, this would be a cyclic

dependency in the data flowing between the groups of M_1 and M_2 as shown in Figure 4.2. Because these mappings are connected, StreamIt optimizes the performance of the subgraphs by compiling all the nodes of each subgraph together to minimize synchronization costs and take advantage of data locality optimization like register allocation. However, in this instance this results in an un-runnable graph because M_1 and M_2 are deadlocked, waiting for data from each other to run. The scheduler interface breaks up M_1 and M_2 into the minimal number of connected subgraphs to break the cyclic dependency using an algorithm to be discussed in 4.2.5, but this now increases the cost of running the subgraph through added synchronization. Instead of having the scheduling algorithm take care of managing when cyclic dependencies need to be broken, the scheduler interface manages it internally. As the scheduling algorithm makes assignments that introduce unique costs to assignment, the interface determines validity, makes necessary changes, updates it's internal costs, and presents the updated cost to the algorithm.

This abstraction is also useful when evaluating the costs of data-flow in the graph as streams are assigned to hardware. Rather than having the scheduling algorithm internally update utilization of edges as nodes are assigned, these costs are also managed by the interface. The interface assigns an edge in the StreamIt graph to a data-flow edge once both of it's associated nodes are assigned to hardware. If this assignment would exceed the constraints of the edge, the node's assignment is rejected.

4.2.3 Interface API

A programmer can also make fine-tuned scheduling commands to the interface before running a scheduling algorithm. The motivation for this feature is for a user to explicitly assign critical parts of their computation to specific hardware units if that stream utilizes that hardware unit more effectively, without having to introduce another set of constraints to the hardware graph and StreamIt graph. An example of this usage is seen in figure 4.3, where the user makes an assignment and the scheduler preserves the assignment while assigning the

rest of the nodes, raising an error if the fine-tuning assignments are infeasible.

```
auto pipe = {
    // Construction of a StreamIt computation graph
    auto matmul = matmul<Tensor, Tensor>();
    pipe->add(matmul);
    // More construction of Streamit graph.
};
Graph G;
pipe->build(G); // Flatten the recursive computation graph
HardwareGraph hwg = genHardwareGraph(); // Generate a hardware layout
SchedulerInterface interface(hwg, G);
interface.assignNode(matmul, interface.getHardwareUnits("GPU")[0]);
ExhaustiveScheduler::schedule(scheduler);
auto assignment = interface.applyAndEmit();
fileCodegen(assignment, "simulator/gen4.c"); // Code generation
```

Figure 4.3: An example of user fine-tuned assignment by assigning `matmul` manually befor e calling the scheduler. Terms not defined in table 4.1 are scheduler `Exhaustive_Scheduler` which has method `void schedule(SchedulingInterface)`. This method is inherited from a an abstract base class `Scheduler` and finds a full assignment of StreamIt nodes to hardware.

4.2.4 Graph Transformations

The scheduling interface exposes a set of commands for schedulers and users to perform graph transformations on the StreamIt graph to fulfill the compiler parallelization operations discussed in 3.1.3. The scheduler interface manages whether nodes can be part of these operations. It's internal record-keeping tracks which nodes have been transformed and how their costs have altered due to transformation.

Only scale-up is supported in the scheduling interface as of this thesis. Whether a node can be scaled is determined by manual labeling as state management for the newest implementation of StreamIt is not fully mature, so detecting filters with state that would dis-allow scaling is currently a challenge. Scaling changes the structure of the *concrete* output graph by functionally changing a node in the *virtual* graph into a duplicate split-join with copies of the original node as children. These copies are not able to be scheduled on the same hardware unit by the scheduler, as that would not result in the increase the parallelism that scaling up aims for.

Type Name	Function Name	Description
`HardwareGraph`	`void addComputeUnit(const HardwareUnit& unit)`	Add a compute unit (node).
	`void addDataflowEdge(const DataflowEdge& edge)`	Add a dataflow edge.
	`const HardwareUnit* findComputeUnit(int id)`	Find compute unit by id.
	`const DataflowEdge* findDataflowEdge(int sourceId, int targetId)`	Find edge unit by ids of source and target nodes.
	`std::vector<DataflowEdge> getEdges(int computeUnitId)`	Return a list of the dataflow edges connected to this node.
`SchedulerInterface`	Constructor	Creates a new instance of `SchedulerInterface`.
	`bool validAssignment(ChipID chip, NodeId node)`	Returns whether the *node* can be assigned to *chip* based on constraints.
	`bool assignNode(ChipID chip, NodeId node)`	Assigns an assigned or unassigned *node* to *chip*.
	`std::vector<NodeID> getUnassigned(HWType type)`	Returns a list of unassigned nodes sorted by descending cost assignable to this hardware *type*, defaults to any type.
	`std::vector<ChipID> getHardwareUnits()`	Returns a list of hardware unit IDs, sorted ascending by utilization of their primary resource.
	`ConstraintMap getCost(NodeID node)`	Returns the cost of all firings of a node during steady state for all hardware types.
	`ConstraintSet getResources(ChipID chip);`	Returns the total resources of the hardware associated *chip* id.
	`ConstraintSet getResourcesRemaining (ChipID chip);`	Returns the total resources after current assignment of the hardware associated *chip* id.
	`std::pair<Graph, HWAssignment> applyAndEmit();`	Returns a concrete StreamIt computation graph and the assignment of nodes to hardware.

Table 4.1: Documentation for the basic methods of `HardwareGraph` and `SchedulerInterface`

Fusing is implicitly part of the the scheduler, as any contiguous part of the StreamIt graph that's mapped to the same hardware unit is compiled as a single executable. Scheduling algorithms take advantage of fusion automatically, unless they produce a hardware mapping where a dependency cycle exists between partitions.

4.2.5 Dependency Breaking Algorithm

Cycles are detected by first constructing a graph where each contiguous group of nodes assigned to the same hardware unit are a single super-node, and directed edges exist between super-nodes where their constituent nodes have directed connections. Running a cycle detection algorithm, such as DFS, on this graph will detect if a cycle exists between a group of super nodes. Next, the nodes of each super-node in the cycle that output into a node in the next super-node of the cycle run DFS with itself as the root; for an example, let's say S_1 have a cycle with S_2 so for each node of S_1 with an output into a node of S_2, run a DFS into S_2. If the DFS finds a node in S_1, then we found at least one cyclic dependency, so we flag the found S_1 node and it's downstream nodes in S_1 to be placed in a new partition. Repeat for all super-nodes in the cycle. Pseudo-code for the algorithm can be found at algorithm 1.

This algorithm does not guarantee optimality of the cut of the partition to break the dependency, but it allows the mapping to be scheduled and for the proper cost of the mapping to be calculated.

4.3 Example Schedulers

In this section we will describe two schedulers implemented on the scheduling interface and the tradeoffs they offer in fitting computation graph to hardware graph constraints.

Algorithm 1 BreakCycles

```
1: procedure BreakCycles(Graph G, HardwareMapping Map)
2:     hwToPartitions ← map from hardware ID to list of partitions
3:     allPartitions ← list of partitions
4:     root ← TopologicalSort[0]
5:     cycles ← true
6:     while True do
7:         superGraph ← empty graph                ▷ New empty graph for partition nodes
8:         for all part ∈ allPartitions do
9:             node ← CreatePartitionNode(part)
10:            AddVertex(superGraph, node)
11:        for all edge ∈ G.edges() do
12:            if edge.from.partition ≠ edge.to.partition then
13:                partitionEdge ← MakeEdgeBetweenPartitions(edge)
14:                AddEdge(superG, partitionEdge)
15:        visited ← tracks visited nodes
16:        curPath ← tracks path in graph for cycle detection
17:        cyclicPartition ← DFSCycleDetect(root.partition, superGraph, visited, curPath)
18:        if cyclicPartition = −1 then No cycles found
19:            break
20:        else
21:            flagged ← DFSDownstream(cyclicPartition, cg)
22:            for all v ∈ flagged do
23:                v.partition ← newPartitionID
24:            Insert(newPartitionID into all_partitions)
25:            Insert(newPartitionID into hwToPartitions using cyclicPartition)
26:    return hwToPartition

27: procedure DFSDownstream(partition, Graph)
28:    visited ← empty set
29:    downstream ← empty set
30:    for all v ∈ PartitionNodes(G, partition) do
31:        for all e ∈ OutEdges(G, v) do      ▷ Only DFS on nodes outside our partition
32:            if partition! = e.to.partition then
33:                FlagDFS(e.to, G, visited, partition, downstream)
       return downstream

34: procedure DFSFlag(v, G, visited, partition, downstream)
35:    if v ∈ visited then
36:        return
37:    visited ← visited ∪ {v}
38:    if v.partition = partition then ▷ Found partition through different partition, cycle!
39:        downstream ← downstream ∪ {v}
40:    for all e ∈ OutEdges(G, v) do
41:        DFSFlag(e.to, G, visited, partition, downstream)
```

4.3.1 Exhaustive Scheduler

The simplest optimal scheduler implemented as a part of this thesis is the exhaustive scheduler. It is functionally a brute force algorithm, where the algorithm combinatorially checks all scheduling possibilities. The algorithm starts by first defining an order to both the streams to be assigned and the hardware units to be assigned to. Then it iterates over the order of streams, assigning each to the first hardware unit in the ordering that is a valid assignment with respect to resources constraints. Once all streams are assigned, the cost of the assignment is calculated, and the assignment is stored if it has the lowest cost so far. Then the algorithm re-assigns the stream of the lowest level to the next hardware unit, repeating until there are no valid hardware units left. The search then backtracks, reassigning the stream in level above and then assigning the bottom-most stream again.

In most abstract terms, let each state of partial assignment of streams to hardware be a node in a graph, with edges connecting each node of a partial assignment up to node n in the order to all nodes that validly assign node n to a hardware unit. This tree is rooted at the assignment of no nodes to hardware, and the leaves are all valid assignments of streams to hardware. Then the exhaustive search performs a DFS on this tree, finding the leave with the lowest cost. Pseudocode for high-level operations of the schedule is Algorithm 2

The tree can be seen to have a branching factor of m from each node where m is the number of hardware units, as the worst case has each node have m different assignments, and a depth of $n + 1$ for the n nodes to be assigned plus the root. A quick analysis of this algorithm yields a runtime of $O(m^n)$, which can quickly get very large but should be done in reasonable amount of time for some small graphs and hardware arrangements.

4.3.2 Heuristic Min-Cut Scheduler

This algorithm attempts to find a graph that minimizes communication while providing a load-balanced set of partitions. This scheduler is a better fit for communication constrained hardware setups, as it will still obey the hardware constraints imposed by the hardware graph,

Algorithm 2 Exhaustive Hardware Assignment Scheduler

```
1: procedure SCHEDULE(scheduler)
2:     nodes ← scheduler.getUnassigned()
3:     units ← scheduler.getHardwareUnits()
4:     bestAssignment ← ∅
5:     bestCost ← ∞
6:     for all possible assignments of nodes to units do        ▷ Backtracking Algorithm.
7:         Apply current assignment to scheduler
8:         if assignment is valid then        ▷ Invalid partial assignments are aborted early.
9:             cost ← scheduler.getRunCost()
10:            if cost < bestCost then
11:                bestCost ← cost
12:                bestAssignment ← current assignment
13:        Revert assignment
14:    for i ← 1 to |nodes| do
15:        Assign nodes[i] to bestAssignment[i] in scheduler
```

but will minimize the cost of edges crossing between hardware mappings. It takes inspiration from prior graph partitioning work [11][12] to iteratively swap nodes that locally reduce the communication cost between partitions while maintaining a balanced load between partitions.

The scheduler begins using a greedy algorithm assign nodes to hardware that prioritizes hardware with low utilization and assigns nodes that would be connected to at least one node already in the partition. Once an initial full assignment is made, a series of passes are made over the nodes to check if they could be swapped with a neighboring node in the StreamIt graph that is in a different partition. For any node, it checks all possible swaps and performs the swap on nodes that produce the best improvement in cost, if any do. Once nodes are swapped, they cannot be swapped for the remainder of the pass. Once t passes occur with no improvement in the cost metric of the graph, the algorithm stops, with t being tunable. Pseudocode can found at Algorithm 3.

Algorithm 3 Heuristic Min-Cut Assignment Scheduler

```
1: procedure SCHEDULE(scheduler)
2:     // Greedy initialization
3:     unassigned ← scheduler.getUnassigned()
4:     while unassigned ≠ ∅ do
5:         units ← scheduler.getHardwareUnits()
6:         for all unit ∈ units do
7:             chip ← compute unit info for unit
8:             candidate ← SELECTNODE(scheduler, unit, chip)
9:             if candidate ≠ null then
10:                scheduler.assignNode(unit, candidate)
11:                break                    ▷ Recompute unassigned after each assignment
12:        unassigned ← scheduler.getUnassigned()
13:    // Opportunistic refinement
14:    noImprovementCount ← 0
15:    maxPassesWithoutImprovement ← t
16:    currentCost ← scheduler.getRunCost()
17:    repeat
18:        frozen ← ∅
19:        improved ← false
20:        for all pairs of nodes (a, b) assigned to different units and not in frozen do
21:            if swapping a and b is valid then
22:                Perform swap temporarily
23:                newCost ← scheduler.getRunCost()
24:                if newCost < currentCost then
25:                    Commit swap
26:                    currentCost ← newCost
27:                    frozen ← frozen ∪ {a, b}
28:                    improved ← true
29:                else
30:                    Revert swap
31:        if not improved then
32:            noImprovementCount ← noImprovementCount + 1
33:        else
34:            noImprovementCount ← 0
35:    until noImprovementCount ≥ maxPassesWithoutImprovement
36: function SELECTNODE(scheduler, unit, chip)
37:    assignedNodes ← scheduler.getAssignment(unit)
38:    if assignedNodes ≠ ∅ then
39:        return heaviest unassigned neighbor of assigned nodes on chip
40:    else
41:        candidates ← top unassigned nodes for chip
42:        return random choice from candidates
```

Chapter 5

Evaluation

In this chapter we evaluate the schedulers on their ability to fit StreamIt graphs to hardware graph constraints on a variety of synthetic graphs of varying structure and steady state schedules. We then test the generated code of these graphs on throughput to evaluate real world performance of the scheduled code.

5.1 Experimental Setup

While the scheduling framework was built for asymmetric hardware setups, at the time of this thesis runtime backends for StreamIt only support pthreads and a monolithic FPGA emulator. Both of these runtimes only allow for uniform hardware targets, so the `HardwareGraph` used for all experiments was that of identically defined `CPU_Core` types fully connected through shared cache. The schedulers were optimizing the same cost objective, which was the load on the most utilized hardware unit added to average utilization of the data-flow edges, formalized below.

$$\text{cost} = \max_{c \in \mathcal{C}} \text{util}(c) + \frac{1}{|\mathcal{E}|} \sum_{e \in \mathcal{E}} \text{comm}(e)$$

As this evaluation mainly serves to illustrate the tradeoffs between the two schedulers in terms of scheduling time and maximizing their cost objectives. We conducted 4 main

benchmarks across a range of core parameters. The benchmarks were chosen to emphasize different extremes of scheduling problems. Long Pipeline and Wide Split-Join are graph that consist of one composing stream with 20 child streams, an allocation of Wide Split-Join is shown in figure 5.1. These test extremes of mapping differing dimensionalities of streams. Regular Pipeline and Increasing Pipeline are to contrast scenarios with roughly uniformly cost nodes against those where nodes differ widely in cost, the latter making proper load-balancing while minimizing communication harder.

5.2 Evaluation

Table 5.1: Scheduler Performance Across Synthetic Benchmarks

Scheduler	**Benchmark**	**Cores**	**Scheduling(s)**	**Runtime(s)**	**Load-balanced**
Exhaustive	Long Pipeline	2	12.5933	3.00381	
Heuristic	Long Pipeline	2	0.00213914	3.00084	
Exhaustive	Long Pipeline	3	Timed Out	N/A	
Heuristic	Long Pipeline	3	0.00253731	3.43068	
Heurisitc	Long Pipeline	8	0.00303133	6.52966	
Exhaustive	Wide Split-Join	2	64.5079	14.5875	
Heuristic	Wide Split-Join	2	0.00239534	14.56	
Exhaustive	Regular Pipeline	4	33.3829	3.25142	
Heuristic	Regular Pipeline	4	0.00259775	4.26053	
Exhaustive	Increasing Pipeline	4	33.9288	24.3808	
Heuristic	Increasing Pipeline	4	0.00264221	54.9757	
Exhaustive	Increasing Pipeline	2	0.013876	35.1631	
Heuristic	Increasing Pipeline	2	0.00264045	48.0592	

As we expect from the exponential runtime analysis, the exhaustive scheduler increases in scheduling time quite rapidly as both the number of nodes and number of cores increase. The Exhaustive and Heuristic schedulers show similar run times across the Long Pipeline and Wide Split-Join benchmarks. When comparing across the differing node cost benchmarks, the Exhaustive scheduler performs much better than Heuristic. We theorize this is because the Heuristic might get caught within a local minima while doing swaps, so it results in a

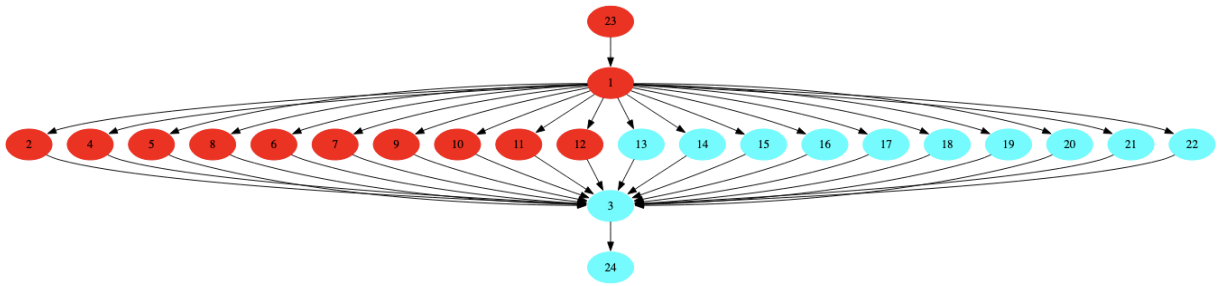
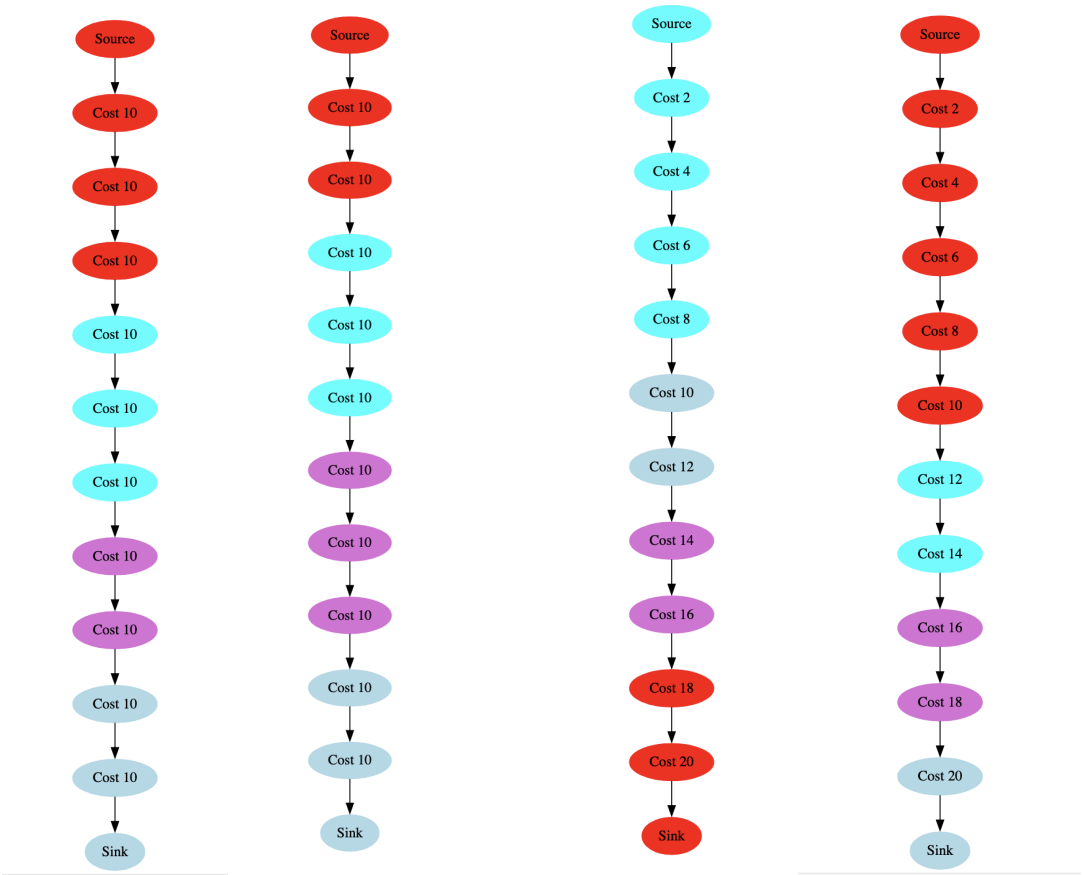


Figure 5.1: Synthetic splitjoin benchmark mapped to two different hardware units.

much worse allocation that searching the entire space.



(a) Regular pipeline benchmark mapped to four hardware units.

(b) Pipeline with nodes of increasing cost mapped to four hardware units.

Figure 5.2: Comparison of two graphs of identical structure with differing node costs produced by both schedulers, left allocation of each figure is the heuristic mapping, and the right is the exhaustive. Equal allocation for equal node costs is intuitive for 5.2a, which the exhaustive gets but the heuristic misses. Figure 5.2b is more nuanced. The hardware could be more load-balanced, but it would introduce more partitions and thus increase synchronization overhead, so the mapping of contiguous but less balanced partitions reduces total cost the most.

Chapter 6

Extending the StreamIt Language with the Split Switch

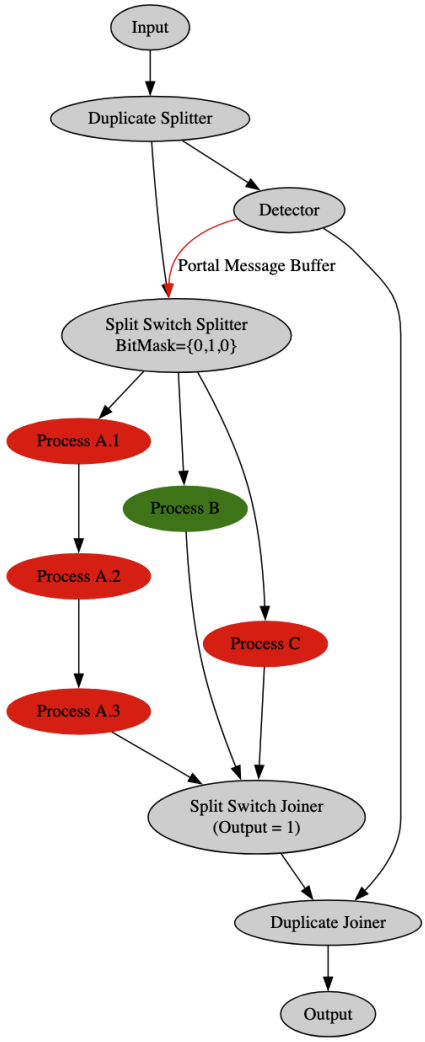
StreamIt is a highly static language, and that is one of it's strengths when performing compiler optimizations. However, if a StreamIt application needs some sub-computations relatively infrequently based on input data, the StreamIt must schedule and continuously run that sub-computation even if the majority of the time it's output is unneeded. Some dynamism in runtime scheduling would allow the application to reduce the computational burden of sub-processes whose execution logic is data-dependent. This motivates the introduction and design of the split-switch, a feature that utilizes StreamIt's portal messaging to dynamically schedule sub-graphs of a StreamIt graph. This section discusses the design of the split-switch and some optimizations implemented for hardware targets optimizing for latency.

```
1  Graph G;
2  auto portal = G.makePortal();
3  auto pipe = pipeline<int, int>([&] (auto p) {
4      auto source = input<int>(1);
5      auto dupsj = duplicateSplitJoin<int, int>([&] (auto dupsj) {
6        auto detect = detector<int, int>(portal);
7        auto sj = splitSwitch<int, int>([&] (auto sj) {
8          sj->addPortal(portal);
9          auto ProcessA = pipeline<int, int>([&] (auto branch_pipe) {
10             branch->add(process<int, int>());
11             branch->add(process<int, int>());
12             branch->add(process<int, int>());
13         }
14         auto ProcessB = process<int, int>();
15         auto ProcessC = process<int, int>();
16         sj->add(1, ProcessA);
17         sj->add(1, ProcessB);
18         sj->add(1, ProcessC);
19       });
20       dupsj->add(1, detect);
21       dupsj->add(1, sj);
22     });
23     auto sink = output<int>();
24     p->add(source);
25     p->add(dupsj);
26     p->add(sink);
27 });
```

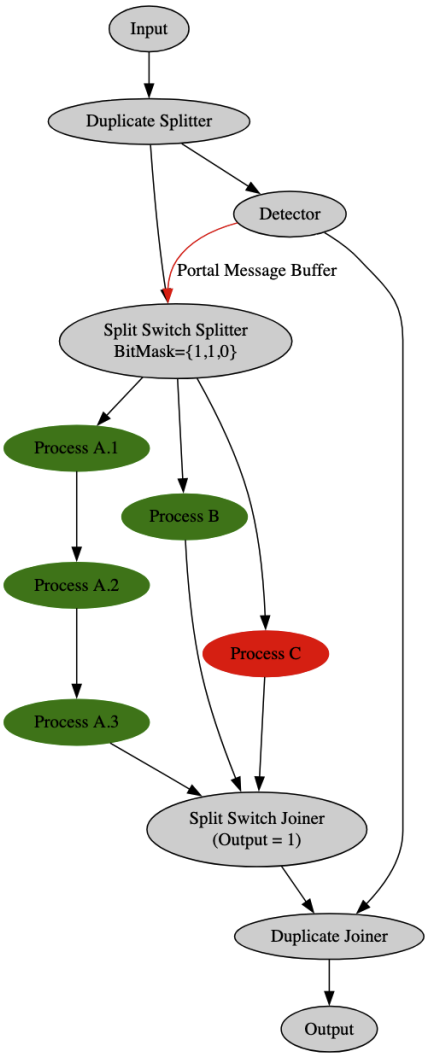
Listing 6.1: StreamIt code to create figure 6.1. `portal` is created on line 2, passed to `detect` on line 6 and to `splitSwitch` on line 8.

6.1 Technical Design

The split switch is essentially a dynamic duplicate split-join. There are some limitations with the data flow rates of the split switch to maintain the necessary static properties of the overall graph. There is a shared input channel, and each child stream must have the same pop rate as in a duplicate split join. Since not all children streams are doing active computation all the time, as otherwise the programmer would not need the split switch, the output rate is not as straight-forward. The output channel push rate can be specified by the number of children streams to always be running; this requirement allows for different data rates out of the split-switch while ensuring there are branches active to supply the data. The output rate



(a) Branch A and it's subgraph is deactivated.



(b) Branch A and it's subgraph is activated.

Figure 6.1: Comparison of the graph before and after partitioning

Figure 6.2: Example of computation graph with a portal. Portal edge is red, between Detector and Split Switch Splitter. In 6.1a branches of the split join are active, seen from the BitMask state and the green color of the active branch. Bitmask entries correspond, in order, to Processes A, B, C. In 6.1b, the branch with the subgraph of Process A is activated. A message that might cause change would be `{01, 11, 00}`; Process A is commanded to switch, Process B is ignored, and Process C is deactivated already.

can also be zero, which can be useful in circumstances where the split switch might have zero active children streams or if children streams output side effects instead of data.

The splitter of the split switch maintains the state of the split switch, which is whether each child stream is currently active. As long as a child stream is active, the splitter will push data to stream's input buffer. This state can be updated via a portal message with `bitMask` datatype. `bitMask` uses two bits to specify each change in state for each of the child streams of the split switch. The first bit acts as an 'ignore' bit, where it being a 1 signals that the split switch does not change the activation of that child stream. The second bit signals the state to change the child stream to, 1 signaling on and 0 signaling off. If the child stream changes state, then the split switch might take architecture specific scheduling commands. An example of a `bitMask` message is shown in 6.1.

6.2 Inlining Optimization

For latency sensitive applications, we've implemented an inlining optimization that pulls the entire sub-graph of the child stream into the same hardware partition as the splitter. The entire branch is then put under an if-statement conditioned on the activity of the branch. This removes synchronization costs, global buffers, and scheduling overhead if a particular part of an application is latency sensitive.

Chapter 7

Conclusion

In this work we developed a new scheduling framework for the StreamIt language tailored to StreamIt's unique scheduling constraints. We implemented and evaluated two schedulers built on top of this frameworks, detailing their tradeoffs across allocation tests specific to StreamIt. Also introduced is a dynamic scheduling feature, SplitSwitch, to reduce the runtime overhead of infrequent computations.

7.1 Future Work

A clear future direction for this work is expanding the scheduling optimizations that the scheduling interface can perform on StreamIt graphs to include vertical fission of nodes.

Another direction that provided some initial motivation for the cost structure of the framework but fell out of scope for this thesis was a scheduling algorithm based in linear optimization. As the constraints are specified for a number of resources, constructing the scheduling problem as as linear optimization problem could provide better schedules than the heuristic scheduler without the runtime of the exhaustive scheduler.

References

[1] P. Tillet, H. T. Kung, and D. Cox. "Triton: an intermediate language and compiler for tiled neural network computations". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 10–19. ISBN: 9781450367196. DOI: 10.1145/3315508.3329973. URL: https://doi.org/10.1145/3315508.3329973.

[2] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. "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. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 519–530. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176. URL: https://doi.org/10.1145/2491956.2462176.

[3] W. Thies. "Language and Compiler Support for Stream Programs". Ph.D. dissertation. Cambridge, MA: Massachusetts Institute of Technology, 2009.

[4] M. I. Gordon. "Compiler techniques for scalable performance of stream programs on multicore architectures". Ph.D. dissertation. Cambridge, MA: Massachusetts Institute of Technology, 2010.

[5] BuildIt-lang contributors. *easy-streamit*. https://github.com/BuildIt-lang/easy-streamit. Accessed: 2025-05-16. 2025.

[6] E. T. Fellheimer. "Dynamic Load-Balancing of StreamIt Cluster Computations". Master's of Engineering. Cambridge, MA: MIT, 2006.

[7] S. Sioutas, S. Stuijk, L. Waeijen, T. Basten, H. Corporaal, and L. Somers. "Schedule Synthesis for Halide Pipelines through Reuse Analysis". *ACM Trans. Archit. Code Optim.* 16.2 (Apr. 2019). ISSN: 1544-3566. DOI: 10.1145/3310248. URL: https://doi.org/10.1145/3310248.

[8] M. Karczmarek, W. Thies, and S. Amarasinghe. "Phased scheduling of stream programs". *SIGPLAN Not.* 38.7 (June 2003), pp. 103–112. ISSN: 0362-1340. DOI: 10.1145/780731.780747. URL: https://doi.org/10.1145/780731.780747.

[9] M. I. Gordon et al. "A stream compiler for communication-exposed architectures". In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS X. San Jose, California: Association for Computing Machinery, 2002, pp. 291–303. ISBN: 1581135742. DOI: 10.1145/605397.605428. URL: https://doi.org/10.1145/605397.605428.

[10] NVIDIA Corporation. *CUDA C++ Programming Guide*. Section: Asynchronous Copy. NVIDIA Corporation. 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#async-copy.

[11] B. W. Kernighan and S. Lin. "An efficient heuristic procedure for partitioning graphs". *The Bell System Technical Journal* 49.2 (1970), pp. 291–307.

[12] C. Fiduccia and R. Mattheyses. "A Linear-Time Heuristic for Improving Network Partitions". In: *19th Design Automation Conference*. 1982, pp. 175–181. DOI: 10.1109/DAC.1982.1585498.