Fast Multistage Compilation of Machine Learning Computation Graphs

by

Kaustubh Dighe

Bachelor of Science in Electrical Engineering and Computer Science,
Massachusetts Institute of Technology, 2023

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 2024

Authored by:	Kaustubh Dighe Department of Electrical Engineering and Computer Science May 17, 2024
Certified by:	Saman Amarasinghe Professor of Electrical Engineering and Computer Science, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair Master of Engineering Thesis Committee

Fast Multistage Compilation of Machine Learning Computation Graphs

by

Kaustubh Dighe

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING
IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Machine learning applications are increasingly requiring fast and more computational power. Many applications like language models have become so large that they are run on distributed systems in parallel. However, getting into the details of optimally scheduling or even just running machine learning models on distributed systems can be a distraction for researchers ideating models. Hence there has been development of abstractions to facilitate running machine learning models in parallel on distributed systems. We present a compiler for the StreamIt language - a language made for abstract signal processing and multicore programming. We use that abstraction as a way to distribute the computation of machine learning models programmed in PyTorch.

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

Acknowledgments

I am grateful to Professor Saman Amarasinghe for brainstorming multiple interesting ideas related to the work in this thesis with me and helping me identify and move forward on this project.

I am very thankful to Ajay Brahmakshatriya, who is a doctoral student. Meetings and discussions with him helped me develop a temperament for research and understand the workflow of a research project. I also highly appreciate the deep technical insights both in the project and also general ideas about compilers and performance engineering I got from him.

I thank my parents for their unwavering support and encouragement during all times. They have a big role in shaping me into what I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the rapidly increasing computation requirements of machine learning workloads, training and deploying them has increasingly required expertise in scheduling workloads on multiple cores and machines. This makes it difficult for people not having an intricate knowledge of the underlying multicore/distributed system to efficiently test or deploy their machine learning models. There is a need for a convenient abstraction that analyzes existing ML programs and convert them to a form that can be easily distributed.

We develop a compiler for the StreamIt language [1], originally meant to optimize streaming applications in the signal processing domain. StreamIt views programs as structured graphs which makes it convenient to schedule different nodes on different cores/machines. Then, we also develop a system that analyzes already developed PyTorch models and generate StreamIt graphs to be fed into the StreamIt compiler.

1.1 The Streaming Abstraction

A lot of applications in signal processing, controls, robotics etc. can be viewed as a continuous time series. There are a lot of significant algorithms like the Fourier transform, and many applications like high resolution video streaming, live sports broadcast, radio transmission and more that lend themselves to a stream abstraction. Going closer to hardware, we can see

that many things like serial communication, input output streams, file streams, UNIX pipes and sockets all use a stream abstraction. Given this, the StreamIt domain-specific language (DSL) was conceptualized 20 years ago [1], [2]. It captures signal processing abstractions like Filters and Pipelines. It also allows for multiple streams of data to join together, or a single stream of data to split into multiple streams. The basic unit of computation is a filter and a stream of data comes in and comes out via channels at possibly different rates from the filters. Multiple filters (and streams in general) may combine in series to form a pipeline.

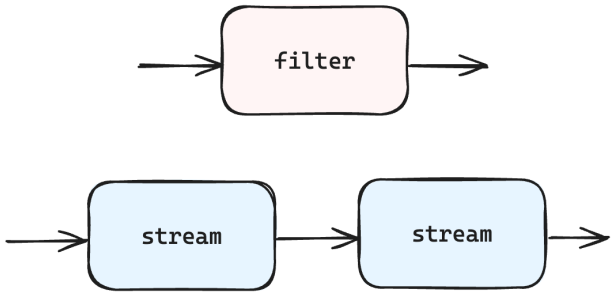

Figure 1.1: Depiction of filters and how multiple streams connect sequentially in a pipeline

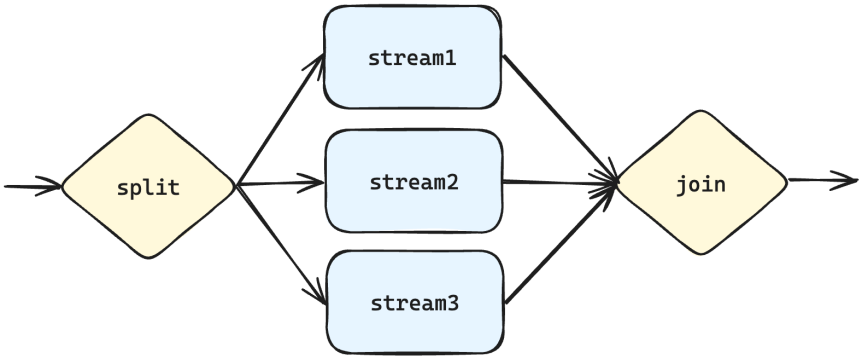

Figure 1.2: Depiction of how streams split into multiple streams, get computed differently, and join back into a single stream

Around the time StreamIt was conceptualized, the idea of just doubling the number of transistors on a computer chip to get performance (Moore's law) was diminishing and computers were increasingly having multiple cores. The abstraction of streams splitting and joining now had another application - splitting computation on multiple cores and synchronizing it by joining. Each stream of data can be split and handled possibly in

parallel. Each such resulting stream can also do different transformations of data. Joins would synchronize these streams into one single stream again. Now, more than two decades later, we are increasingly using distributed systems for computations and data storage. Our data and computation needs have become so large that a single machine is often too weak to handle it. This same streaming abstraction can be used to denote communication over distributed systems.

1.2 Multistage Compilation and BuildIt

Multistage compilation is to have programs that get compiled in multiple stages, taking some inputs in each stage. Expressions and language constructs take concrete shapes at various stages of compilation. This allows us to generate efficient code specialized for various input conditions. As a very simple example, the code to find the exponent of a base would look like this in single-stage compiled C/C++.

```
int power(int base, int exp) {
    int res = 1;
    while (exp > 0) {
        if (exp % 2 == 1) res = res * base;
        res = res * res;
        exp = exp / 2;
    }
    return res;
}
```

Figure 1.3: Simple C code to find exponent

Notice here that there are multiple conditional statements and loop all of which makes the execution slower. However, this is the only form that is the most general and easy to read. Multi-staging can allow our code to take the exponent in the first stage and generate a code free from recursive calls, loops and conditionals in the second stage. This makes the code that actually runs faster. This idea can be extended to more complicated domains where some properties of the inputs are checked in the first stage, leaving us with a much

simpler and faster code to run.

To perform multi-staging, we use BuildIt [3] which is a library that introduced mainly two template wrapper classes - `static_var<T>` and `dyn_var<T>`. The first kind is that of static variables. Static variables take some concrete values in the first stage. Many control structures also change based on these variables. For instance, if the iterator of a loop is a static variable, the entire loop body will be unrolled. Dynamic variables on the other hand are walking abstract syntax trees (ASTs). They generate the code for next stage based on the ASTs. Here is an example of how the exponent example is written in BuildIt.

```
dyn_var<int> power(dyn_var<int> base, static_var<int> exp) {
    dyn_var<int> res=1;
    while (exp > 0) {
        if (exp % 2 == 1) res = res * base;
        res = res * res;
        exp = exp / 2;
    }
    return res;
}
```

Figure 1.4: BuildIt first stage code to find exponent

Notice how few changes needed to be made. Also note that we make the base and return values dynamic, and the exponent a static variable. This means that we are making specialized functions for each power. Here is what code will be generated for the next stage if we set the exponent to 5.

```
int power_5(int arg0) {
    int var1 = arg0;
    int var2 = 1;
    var2 = var2 * var1;
    var2 = var2 * var2;
    var2 = var2 * var2;
    var2 = var2 * var1;
    var2 = var2 * var2;
    return var2;
}
```

Figure 1.5: BuildIt 2nd stage code for exp = 5

Since BuildIt is a library, it cannot read within the language syntax and figure out what

variable names we have chosen and just assigns names, but we can see that the loop has been unrolled and all conditions on the exponent have been removed. The next stage of compilation will be done by the regular C compiler on a user's machine that generates the assembly code to be run. This example just showed a 2 stage compilation process, but it can be extended to multiple nested dynamic variables. Each nesting of dynamic around a variable gets processed at each stage of compilation leading to interesting patterns based on the static inputs at each stage.

1.3 Using BuildIt for Compilers

As we saw in the previous section, multistage compilation is a very powerful concept. We now discuss some general benefits of using that power to quickly and easily build compilers. Typically compilers are huge and extremely complicated pieces of software running into tens and even hundreds of thousands of lines of code [4]. The code is first viewed as a string of characters and parsed based on the syntax of the language. Typically parse trees are generated from this and syntax errors (like missing a semicolon or not closing a scope or indentation errors) are caught at this stage. Then further passes over this tree check for semantic errors (like adding ints to strings or calling a function that does not exist.) Then the control flow (how the execution of lines of code might happen in order, after considering conditional statements, loops and function calls) of this code is encoded in an intermediate representation or a control flow graph. This graph is enough to generate the final assembly code. However, many optimization passes are done over the control flow graph to generate much faster assembly code by removing redundant code or by exploiting some patterns like vectorization of loops. This requires very complex analysis of the nature of the code. This is a lot of work.

Developers of domain specific languages who just need a small language to run really fast for a specific domain neither have the time and resources to code all this for each new

language nor the expertise. They often make libraries in existing popular languages, that generate optimal code. But still that just removes the effort of parsing some text. All the other steps - intermediate representation, optimizations, code generation remain. Here is where multistage compilation comes to the rescue. As we saw in the example of the previous section, based on inputs of one stage, a very different and efficient code is generated in the next stage. In many cases, control structures change as well like loops being unrolled, structs and classes being eliminated into simple linear code, nested loops being generated from a single level loop and so on. Considering all this, developers have the following advantages while making compilers using multistage programming:

- The code to be written in the first stage looks very similar to what one would write if one were to just write a library. So, compiler developers can just write a library and only the domain specific optimizations.

- The other optimizations that are not domain specific will automatically be applied when the final stage code gets compiled by a highly optimized off-the-shelf compiler like LLVM or GCC.

- Many domain specific optimizations involve checking inputs for some special properties like tensors being sparse or dense (to consider applying sparse tensor representations), multiple arrays not overlapping in memory (to correctly vectorize loops) and many more. All these checks can happen at the first stage itself and as we saw in the example of exponentiation, all conditional checks will get evaluated in the first stage itself and the final code to run will be specialized for those parameters.

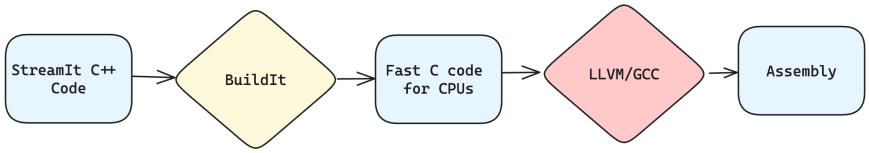

Figure 1.6: How StreamIt Compiler generates code in two stages

Considering these compelling reasons, we decided to develop the compiler for StreamIt language on top of BuildIt in C++. BuildIt provides the following additional advantages:

- BuildIt has multiple backends that generate efficient C code for CPUs and CUDA code for Nvidia GPUs [3], [5]. This allows us to write a single code for StreamIt compiler and compile it for multiple backends.
- Debugging tools were developed for BuildIt [6] and we get access to that.

1.4 Machine Learning and StreamIt

As we said while introducing the streaming abstraction, computation of any kind can also be thought of as a stream of data going through multiple instructions and getting transformed. The set of filters and their dependence on one another creates a computation graph. Machine learning workloads can hence also be visualized as streams of tensors going through a series of operations. Filters can be operations like neural network layers, activations, while tensors can be split when the tensors are too large to be processed at once and we want to distribute over multiple cores or machines. Thus, this fits right as an application of StreamIt. Popular machine learning libraries like PyTorch [7] and TensorFlow [8], also create a computation graph of the operations written in python and then run C code for that. However, these graphs are dynamic in nature as they are created as the python code is interpreted and can change based on input. However, StreamIt graphs are static, that is irrespective of the input, we know the size of the graph at compile time. We implement the conversion of PyTorch computation graphs into StreamIt graphs via an intermediate representation and study some edge cases of it.

1.5 Contributions

- I built the StreamIt compiler in a multistaged manner with BuildIt as described in detail in Chapter 2.
- Compiling PyTorch graphs into StreamIt streams as described in detail in Chapter 4.

Chapter 2

The StreamIt Compiler

2.1 Effect of BuildIt on Structs and Classes

Before jumping into the details of StreamIt language and its compiler, let us see a quick example of how BuildIt handles a small library having classes, inheritance and virtual functions and generates very simple and hence fast code.

```
1  struct Number {
2      virtual ~Number() {}
3      virtual void scale(dyn_var<int> factor) {}
4  };
5
6  struct Complex : public Number {
7      dyn_var<int> re, im;
8
9      Complex(dyn_var<int> re, dyn_var<int> im) {
10         this->re = re;
11         this->im = im;
12     }
13
14     void scale(dyn_var<int> factor) {
15         this->re *= factor;
16         this->im *= factor;
17     }
18 };
```

Figure 2.1: Complex number library example code

In 2.1, we have an example of an abstract class Number that has a virtual function scale,

which multiplies that number by a factor. We see a derived class for complex numbers. Below in 2.2, we have the very simple program that we run.

```
std::shared_ptr<Number> number = std::make_shared<Complex>(5, 6);
number->scale(4);
```

Figure 2.2: Calling the function scale on a complex number

The code generated by BuildIt is as follows. Notice how structs, inheritance, function calls, virtual functions all collapse into the simple operation that actually happens. Having objects and virtual function calls have overheads which get saved in this manner without compromising on writing modular object-oriented libraries.

```
int var1 = 6;
int var2;
int var3;
var2 = 5;
var3 = var1;
int var4 = 4;
var2 = var2 * var4;
var3 = var3 * var4;
```

Figure 2.3: BuildIt generated code for calling scale on complex number

One can see in the snippet in 2.3, var2 and var3 are the real and imaginary parts of the complex number, var4 is the scaling factor. I ran this snippet a 1000 times as well as a version which is just a library using O3 compiler optimizations. That is, from 2.1, I removed dynamic variables and made them regular ints. Then running the code in 2.2 is the same as running a library where objects and inheritance has not collapsed into a simpler form. The call to scale is a virtual function call. The BuildIt generated code took almost no time (42 nanoseconds) for 1000 runs while the version without BuildIt took 108625 nanoseconds. The C++ compiler was easily able to optimize the code in 2.3, using constant folding and common subexpression elimination, but the library call could not be optimized away and was way slower. This elimination of overheads like virtual functions, dynamic dispatch, function calls, conditional statements etc. is why code generation is necessary.

2.2 The StreamIt Language - Basic Types

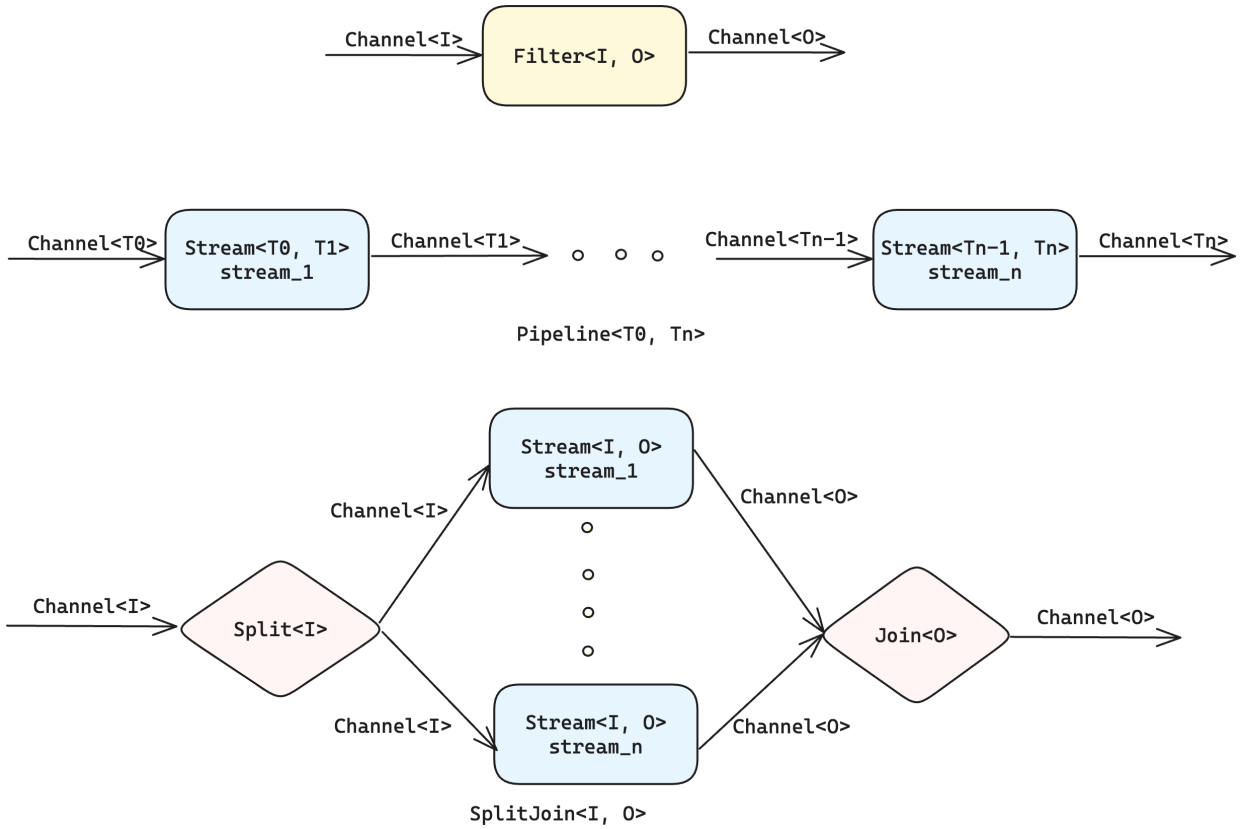

Figure 2.4: Detailed Figure of StreamIt types

As introduced in [1] and discussed in Chapter 1, the StreamIt language is based on streams of data. I have made a C++ compiler for the StreamIt language using BuildIt multistage compilation. A `Stream` is the top level abstract class which has a stream of data as an input, which gets processed and transformed and pushed into an output stream of data. These input and output streams are called `Channel`s. A `Stream` could be just a stand-alone `Filter`, a `Pipeline` of multiple `Stream`s connected in series, or a `SplitJoin` which involves splitting an input stream into multiple parts or duplicating some parts of the stream and processing each part separately. Notice that any filter can change the type of input as well, and hence by extension, any stream can have different input and output types. Hence, we make all these types templates as `Stream<I, O>` where I is the input type and O is the

output type. Since `Channels` are just streams of data, they have only the type of data as a template parameter - `Channel<T>`. These types are visualized in 2.4

2.3 Work Function and Flow Rates

Each `Stream<I, O>` object has a `work()` function in it. Computation happens and data moves forward in the stream when this function is called. The only place where the user specifies the work function is the filter. The work functions for other types of streams are given by the language as will be discussed in a short while. Each `Stream<I, O>` also has a push rate, a peek rate and a pop rate.

- Pop rate is the number of elements from the input channel that get popped from it in each call to the work function.
- Peek rate is the number of elements from the input channel that are accessed in each call to the work function. Notice here that these elements are just accessed and possibly not removed from the input stream.
- Push rate is the number of elements that each call to work function generates as an output that gets pushed into the output channel.

At this point things will be clearer with examples. If we want to add pairs of numbers in a stream, we make a filter as follows

```
class Adder : public Filter <int, int> {
public:
  Adder() : Filter <int, int>(2, 2, 1) {}
  void work() {
    this->push(this->pop() + this->pop());
  }
};
```

Figure 2.5: Adder Filter code snippet

The functionality of the filter is implemented by the user. The Filter class is constructed with arguments 2, 2 and 1. This means that the pop rate and peek rate is 2 and the push rate is 1. This is because in each call to `work()`, 2 elements are popped from the input channel, they are added together to generate only one element which is pushed into the output channel. Here, the peek rate is also 2 because 2 elements are accessed while they were popped. An example where the peek rate is different from the pop rate would be finding the moving average of the last `window` elements of the input stream.

```
class MovingAverage : public Filter <float, float> {
  int window;
public:
  MovingAverage(int window) : Filter <float, float>(1, window, 1) {
    this->window = window;
  }
  void work() {
    float sum = 0.0;
    for(int i = 0; i < window; i++) {
      sum += this->peek(i); // peek the ith most recent element in the stream
    }
    this->push(sum / window);
    this->pop(); // move the stream forward
  }
};
```

Figure 2.6: Moving Average Filter code snippet

In this example, notice that we are accessing `window` elements in each call to the work function, but we are only moving the input channel forward by 1 element by popping only once. This is because we want moving average at each data point, but for computing the average we need to read previous `window` elements. The pop rate is 1 because the output at each call is one value – the average. Also, note how 2.5, 2.6 look so much like implementing a library, but when the work function of class `Filter<I, O>` will get called, BuildIt will generate C code that gets rid of all the classes, virtual functions and inheritance.

2.4 Pipelines - Connecting Filters Serially

We saw how individual filters are implemented by the programmers who use StreamIt language. These filters are then connected to each other via `Pipeline`s and `SplitJoin`s. During the design of this, I had to make some choices. I have to support that each stream connected in a pipeline can have any input and output types as long as the output type of a stream is the same as the input type of the next stream. For this, there is no easy way in C++ to just create an array or a vector of general stream objects because it is just a template. When different types are put in those templates, they are different types and we cannot make an array or vector out of it. So I decided on the following design. An object of type `Pipeline<I, O>` can be one of the two -

- `OneElementPipeline<I, O>(Stream<I, O> *stream)` This class is constructed from a single stream. It takes in a pointer to that as an argument to the constructor.
- `MultipleElementPipeline<I, O>(Pipeline<I, O1> *first, Stream<O1, O2> *last)` This class essentially appends the stream `last` into an existing pipeline `first` to create a new pipeline. This makes the series of streams into a linked list like structure.

The work function for this design was implemented as follows -

- For the one element pipeline, just call the work function of the underlying stream but wait until there are enough elements to satisfy the pop and peek rates of that stream. This was done by checking the size of the input channel.
- For the multiple element pipeline, call the work function of the first which is a pipeline (this will eventually call individual stream work functions,) and then call the work function of the last stream. While doing this, we ensure at each step that there are enough elements to satisfy the pop and peek rates of the upcoming stream.

This is how one can join two filters to form a pipeline.

```
std::shared_ptr<Pipeline<int, int>> p =  add<int, int, int>(
    pipeline<int, int>(std::make_shared<Adder<int>>(3)),
    std::make_shared<Adder<int>>(-1)
);
```

Figure 2.7: Making Linked List like Pipeline

Note that the adders mentioned in this code are not the adders from 2.5 which added two numbers from the data itself. These adders add the number given in their constructor argument to each element of the input data (push rate = pop rate = peek rate = 1.) And, the compiler generates the code given in 2.8 for this. The reader can observe that as expected all classes, inheritances, function calls, constructors etc. have disappeared and we are saving that overhead. However, notice how the code checks for the size of the deques (which are the input and output channels,) in a while loop. This is a very dynamic and inefficient model which defeats the purpose of a compiler. Other shortcomings are –

- Creating long pipelines is inconvenient because we have to keep creating `MultipleElementPipeline`s for every pair.
- While traversing the pipeline by writing a recursive code is very simple, but it is inefficient. The first stream in this pipeline is buried under several wrappers of `MultipleElementPipeline`s. This means that each wrapper's work function adds a layer of check on the input size before calling work function of the first stream. Also, these checks cannot be eliminated by BuildIt as it is dependent on the input stream at runtime.
- One key aspect of StreamIt is generating a highly optimized static graph at compile time. This is the reason for including push, pop and peek rates at compile time so that the compiler does away with constantly checking for availability of data. This linked list like structure generates a dynamic code which does not exploit the fact that we know the push, pop and peek rates at compile time.

```
void function (std::istream& arg0, std::ostream& arg1) {
  int var5 = 3;
  int var9 = -1;
  std::deque<int> var11;
  std::deque<int> var12;
  int var13 = 3;
  std::deque<int> var18;
  int var19 = -1;
  int var24;
  int var25 = 0;
  while (arg0 >> var24) {
    var11.push_back(var24);
    while (1) {
      int var28 = var11.size();
      if (var28 >= 1) {
        int var29 = var11.front();
        var11.pop_front();
        var12.push_back(var29 + var13);
      } else {
        break;
      }
    }
    while (1) {
      int var32 = var12.size();
      if (var32 >= 1) {
        int var33 = var12.front();
        var12.pop_front();
        var18.push_back(var33 + var19);
      } else {
        break;
      }
    }
    var25 = var25 + 1;
    if (var25 >= 1) {
      while (var25) {
        int var36 = var18.front();
        var18.pop_front();
        arg1 << var36;
        arg1 << "\n";
        var25 = var25 - 1;
      }
    }
  }
}
```

Figure 2.8: Output of Fig 2.7

- I had to make the compiler ready for future optimization passes. Being able to quickly iterate over the entire array of streams was essential for this. A linked list like structure was a hindrance.

Therefore, I decided to make a general `StreamContainer` class from which the template classes including `Stream<I, O>` inherit. I then made a new kind of pipeline called `VectorPipeline<I, O>`, which has a vector of stream containers. This is also equally general as the previous design with regards to allowing streams to have any input output types as well as , but allows us to create vectors of stream containers. So now the inheritance chain looks like this

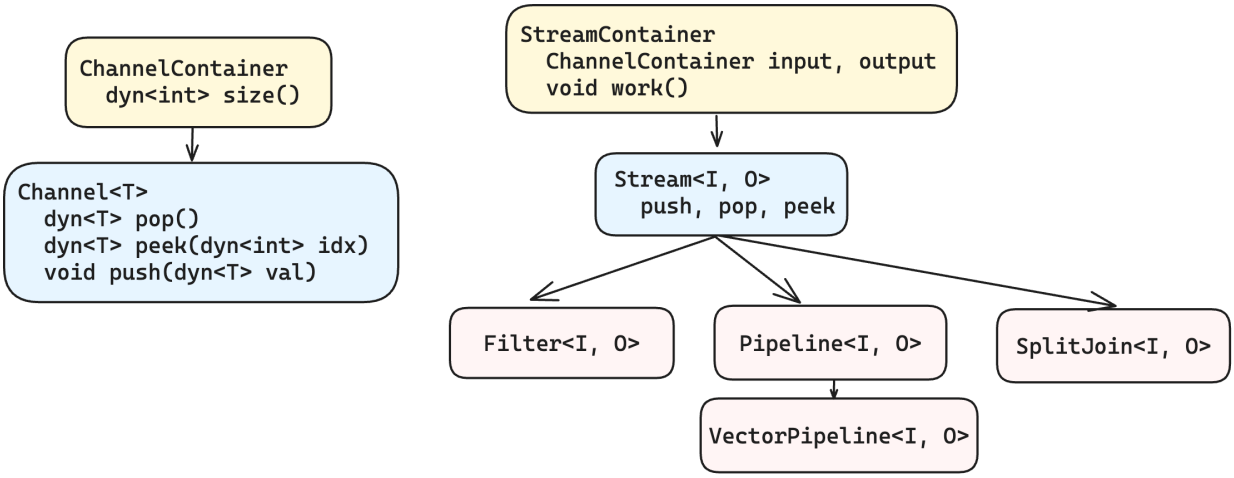

Figure 2.9: A tree showing all the streamIt types and their inheritance heirarchy

This allows for easy access to each stream within the pipeline. A dynamic way to implement the work function would still be to just iterate over all streams, call their work functions after checking if enough elements are there to call the work function. Also, to further simplify things for this dynamic code generation, I noticed that we have to keep checking if enough elements are there to call the work function of a stream (which eventually ends up in a filter.) Hence, I made a wrapper work function for the `Filter<I, O>` that calls the user specified work function after checking for enough elements in the input channel for peeking and popping elements. This simplifies the work function of `VectorPipeline<I, O>`

by just iterating over all streams. Notice how we use a static iterator. This means that in the generated code, this entire loop will be unrolled.

```
void work() {
    for(builder::static_var<size_t> i = 0; i < streams.size(); i++) {
        streams[i]->work();
    }
}
```

Figure 2.10: Work function of VectorPipeline<I, O>

2.5 SplitJoins - Connecting Streams in Parallel

Now that we know how to have filters connected in series to create pipelines, let's see how we can connect these streams in parallel. For a parallel connection, we need to figure out ways to split the input channel of data and ways to join multiple streams of data into one output channel. Since we plan to use StreamIt for parallelizing programs, let's think of ways in which we want parallelism -

- The same data needs to be computed in multiple ways in parallel and combined in some way. For example, we have a value `x`, and we want to find `f(x) + g(x)`. Notice that the same data point is going into both functions, but we want `f` and `g` to be computed in parallel. This means while splitting the stream we need to create extra copies of each data point. While synchronizing (joining) the outputs, we want one point from the output of `f` and one point from `g` and add them.

- Different parts of data go to different parallel streams. For example, if we are multiplying a vector by a constant scalar, this can be parallelized by splitting the vector into multiple streams. However, notice here that we do not need to create extra copies, we just need to send parts of the vector to various streams and join the results back in the same order.

- In the example of the previous case, it is possible that we want to load balance in such

a way that one stream has a higher amount of data going through it as compared to another. Then while joining also we need to take care of this imbalance.

StreamIt is very powerful in allowing a very general framework for this purpose. It provides for the following types of splitters which I created by making derived classes from a general class `Split`.

- `DuplicateSplitter(int num_split)` This class has a parameter that tells us how many output streams we want to duplicate the input into. Elements are popped one by one from the input channel and a copy of it is pushed into each of the `num_split` output channels in each call to the work function.

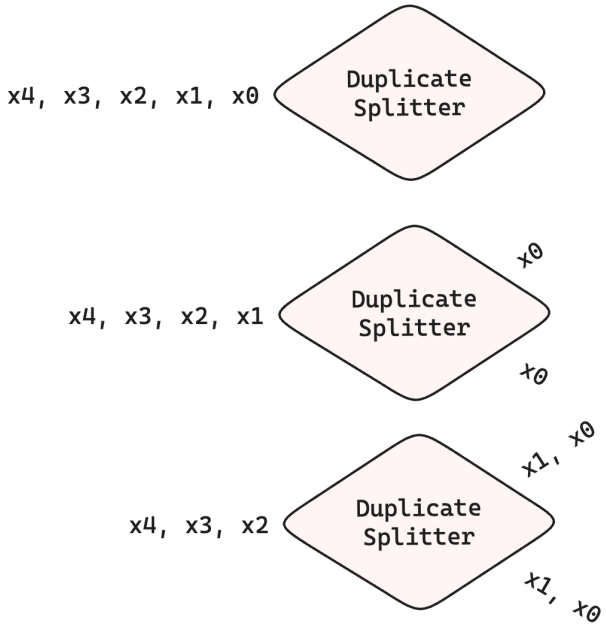

Figure 2.11: Duplicate splitter in action with num_split = 2

- `RoundRobinSplitter(vector<int> &weights)` This class is for splitting the input channel into multiple output channel. The argument `weights` gives us the number of elements to be pushed to each output channel in each call to the work function. This way, we are not constrained to send an equal amount of data to each stream. Look at the figure below to see this in action.

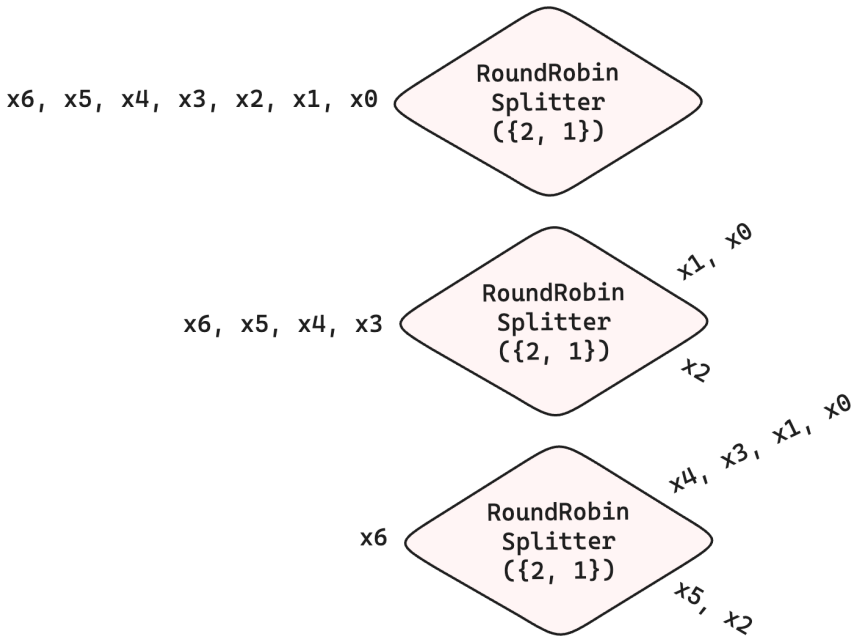

Figure 2.12: Round Robin splitter in action with weights = {2, 1}

Notice how `weights = {2, 1}` implies that the first 2 elements will be sent to the first stream, and the third to the second stream during each call to the work function.

For joining, since in general we cannot guarantee that for the same input, each stream will generate the same output (or even the same number of outputs,) we do not have a duplicate joiner. We just have a `RoundRobinJoiner(vector<int> &weights)`. The weights function in the same manner as the splitter. However, notice here that the weights of a joiner may be different from the weights of a splitter. This makes the StreamIt language even more general and powerful to express more complicated parallel settings. A `SplitJoin` is created by a splitter, multiple streams and a joiner. The work function for a `SplitJoin` involves calling the work function of the splitter, to pop data from the input channel and distribute it to the streams. Then call work functions on each stream in parallel and finally, call work on the joiner to join everything back into one output channel.

2.6 Static Scheduling

As we saw in 2.8, a dynamic schedule of StreamIt filters creates an inefficient and complex code. To address this issue, especially in steady state, we can take the least common multiple of all the pop rates, peek rates and push rates and in one go, we extract that amount of data and push it through the stream.

2.6.1 Flow Constraints in Pipelines

As an example, suppose a pipeline has 2 stream. The first stream S_1 has pop, peek and push rates of 2, 3 and 5 respectively and the second stream S_2 has pop, peek and push rates of 4, 4 and 7 respectively. For both streams, we can just consider the peek rate which is the maximum of pop and peek rates and that is the amount of data points that need to be present in the input channel to successfully execute the work function of a stream. So, we have FS_1 needing 3 values and generating 5 values and S_2 needs 4 values and generates 2 values. Thus, if S_1 takes in $3k$ values, it generates $5k$ values for some positive integer k. Then, S_2 gets these $5k$ values and generates $5k \times 7/4$ values. Also, $5k$ needs to be divisible by 4. If we set $k = 4$, it solves the problem. In one go, S_1 takes 12 values and generates 20 which means S_2 can run its work function 5 times generating 35 values as output. Effectively, the key constraint here was that S_1 should generate such a number of values that an integral number of work function calls can be executed by S_2. Extend this to n streams, and let the stream i $(i = 1, .., n)$ having a peek rate of $peek_i$ and push rate of $push_i$. The pop rates do not matter as the peek rates are always going to be at least as much as the pop rates and we need peek rate amount of data to execute a work function. For $n = 2$, we saw that $LCM(push_1, peek_2)$ amount of data is generated by S_1 and absorbed by S_2.

Adding another stream S_3, if S_2 absorbs $LCM(push_1, peek_2)$, it generates $\frac{LCM(push_1, peek_2) \times push_2}{peek_2}$ values and that should be an integral multiple of $peek_3$. That means, $LCM(push_1, peek_2) \times push_2$ needs to be divisible by $peek_2 \times peek_3$. For simplicity, if we

have $l = push_1 \times peek_2 \times peek_3$ amount of values generated by S_1, then that will satisfy both S_2 and S_3's. Extending it to a general $n \geq 2$, we need $l_n = push_1 \times peek_2 \times peek_3 \times .. \times peek_n$ values being generated by S_1 in each bunch to satisfy the peek rates of all streams in the pipeline. Notice that only the peek rates are important because they get divided, while push rates are multiplied to the flow rate flowing through the pipeline. Effectively, we can say that the net peek rate of the entire stream is $peek_{1:n} = \frac{l_n * peek_1}{push_1} = \prod_{i=1}^{n} peek_i$ and the net push rate is $push_{1:n} = \prod_{i=1}^{n} push_i$. The pop rate would be the number of times the first stream is run times its pop rate. $pop_{1:n} = \frac{peek_{1:n} \times pop_1}{peek_1}$.

$$peek_{pipeline} = \prod_{stream \in pipeline} peek_{stream} \tag{2.1}$$

$$push_{pipeline} = \prod_{stream \in pipeline} push_{stream} \tag{2.2}$$

$$pop_{pipeline} = \frac{peek_{pipeline} \times pop_1}{peek_1} \tag{2.3}$$

2.6.2 Flow Constraints in Split Joins

If we are in a duplicate split join with streams 1 to n, then if k elements are popped by the splitter, each stream gets those same k elements, and k is the pop and peek rate for this split join. That means k has to be a multiple of the peek rates of all the streams. Hence the i^{th} stream will generate $l_i = \frac{k \times push_i}{peek_i}$ elements as the output. Now, let the weight of the round robin joiner for this stream is w_i. Let the joiner perform m sweeps of interleaving outputs from all streams. If the stream S_i is run n_i times ($n_i = \frac{k}{peek_i}$,) then $n_i push_i = m w_i$. Notice here that since n_i are always in the inverse ratio of the peek rates, and $push_i$ are also known, we cannot have arbitrary weights for joining. Assuming that w_i are valid, that is, there exists a positive integer m that satisfies given conditions, $\frac{k push_i}{w_i peek_i} = m$ for all i. Thus,

k also needs to be divisible by $w_i peek_i$ for all i.

$$peek_{dSJ} = pop_{dSJ} = LCM(\{w_i \times peek_i\}_{i=1}^{n}) \tag{2.4}$$

The push rate is the number of sweeps of joiner times number of elements output in each sweep.

$$push_{dSJ} = \frac{peek_{dSJ} push_1}{peek_1} \sum_{i=1}^{n} w_i \tag{2.5}$$

Using the mentioned equations, we can construct static schedules by executing work functions of individual filters multiple times at once. This was implemented by the research group. Another optimization to be done as future work is not to use explicit deque channels for input and output but just allocate a global array buffer for the entire stream, and then the channels keep track of the indices while pushing, peeking and popping.

Chapter 3

Machine Learning Computation

Popular machine leraning libraries such as PyTorch [7] and TensorFlow [8] are examples of embedded domain specific languages. They are embedded as a library within Python, but they generate a computation graph from the Python code we write. The underlying structures are in C. Several optimizations are done on this graph and also within individual functions. There are a lot of optimizations like vectorization of loops, efficient storage of tensors, avoiding copies of tensors when it can be avoided and many more.

3.1 Computation Graphs

Both PyTorch and TensorFlow create computation graphs to internally store the computation described in Python [9], [10]. As given in [10], a key application of this graph is to automatically generate a reverse graph for automatic differentiation, as shown in the figure below taken directly from that website. There are two ways in which these graphs can be created - static and dynamic. Earlier versions of TensorFlow had only static computation graphs. This is where first the programmer explicitly creates a graph, allocates space for tensors and then runs it whenever it is required. See 3.2.

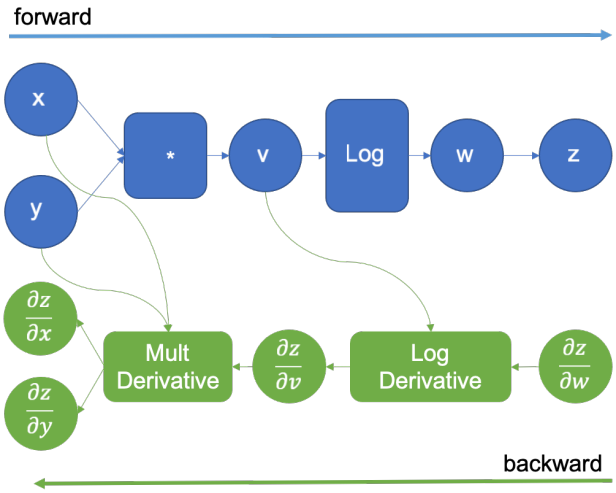

Figure 3.1: Computation graph of $z = log(xy)$ with auto differentiation. (Credit: pytorch.org)

```
import tensorflow as tf

matrix1 = tf.placeholder(tf.float32, shape=(2, 3), name='matrix1')
matrix2 = tf.placeholder(tf.float32, shape=(3, 2), name='matrix2')

product = tf.matmul(matrix1, matrix2)

matrix1_data = [[1, 2, 3], [4, 5, 6]]
matrix2_data = [[7, 8], [9, 10], [11, 12]]

# Create a session to run the computation
with tf.Session() as sess:
    # Run the session to compute the product
    result = sess.run(product, feed_dict={matrix1: matrix1_data, matrix2:
    matrix2_data})
    print(result)
```

Figure 3.2: Example of TensorFlow Static Graph Creation

The main advantage of a static computation graph is that all sizes and operations are known before runtime and nothing is dependent on the input. This means that optimization passes can be made over the entire graph before running it. As listed in [11], TensorFlow graphs go through a lot of graph optimization passes like constant folding, eliminating common subexpressions, pruning nodes, parallelizing nodes and more. This means that static computation graphs have an advantage where resources are scarce or when we value speed.

PyTorch on the other hand, along with newer versions of Tensorflow, have a dynamic computation graph creation. This allows a lot of flexibility on size and shape of the computation graph as it is created on the fly. Pytorch model classes specify forward function where we can write operations, but they can be recursive or have any input (no placeholder to specify input size). Many research and production models need to have inputs of any size and that increases accuracy.

- Convolutional Neural Networks (CNNs) may need to process images of multiple sizes and padding or downsizing leads to poor results.

- Sequence models such as those in natural language processing or signal processing used to pad all sentences/sequences in a dataset to the maximum sequence length. This was not only wasteful but also resulted in poor results.

- An interesting edge case is that of the Tree-Recursive Neural Networks. These are a new type of RNNs which understand sentence grammar in a tree fashion, connecting grammatically related words together, building phrases etc. This means that the tree can have a different structure for each input. These are also used in other domains of structure prediction e.g. making robots understand tasks and their subtasks etc. This is very easy to create in a dynamic graph [12].

Also, since the computation graph creation happens implicitly, the programmer just gets the experience of writing in Python rather than like writing in a new language. However, dynamic graphs do not give opportunities for static type checking and doing optimization passes before runtime. The industry opinion seems to be favoring dynamic graphs since last many years because of the ease of use of PyTorch and PyTorch is also significantly optimized. Hence recent versions of TensorFlow also adapt a similar approach. We saw an opportunity here to bring all the advantages of an arguably better static graph system StreamIt with a user interface of PyTorch. We will talk about that in the next chapter.

3.2 Related Work for Distributed ML Execution

A lot of work in industry and academia has been done in trying to figure out good abstractions of making machine learning workloads faster on single nodes, but Yadav et. al. mention in [13] that not much has been done to create powerful abstractions over . The Distributed Tensor Algebra Compiler (DISTAL) introduced in [13], is a very recent work in this domain and a addresses the problem in a complete manner. The paper addresses this problem by extending the Tensor Algebra Compiler (TACO) [14] to run on distributed supercomputers. It provides a very python-like language (easy to understand and write) for three purposes

- Describing the storage of tensors (data types, sizes, place in memory). For this they also describe the grid of the supercomputer machine.
- A scheduling language to describe how the computation is to be scheduled over the machine.
- Describing the tensor algebra kernel (a computation involving tensors, similar to machine learning layers).

Using simple functions like `distribute` and `communicate`, this system allows programmers to distribute workloads over a specified set of machines and communicate results between machines. Distribution happens by evaluating the iteration space by considering broadcasting dimensions and then it is divided over the given devices. They have achieved a speed improvement of 1.8x to 3.7x over 256 nodes of Lassen supercomputer on tensor multiplications.

A more detailed library for distributed training is provided by PyTorch itself - the `torch.distributed` package [15]. This allows a more fine grained control over how things are distributed. There is a Remote Procedure Call library which can be used to initialize and communicate between machines. Additionally, abstractions such as `DistributedDataParallel`

and `FullyShardedDataParallel` allow for single machine multiple GPU and multiple machine distributed training systems respectively [15]. [16] introduces detailed evaluation of `DistributedDataParallel` and its performance speedups over existing distributed frameworks while training benchmark models such as ResNet and BERT.

The work presented in this thesis aims to take such related work to the next level by allowing users to write a simple PyTorch model as they would do while researching the model, then allow them to compile that model down to StreamIt and use StreamIt scheduling to distribute and parallelize the execution. A key part of this is to compile PyTorch graphs into StreamIt streams and then schedule those streams.

Chapter 4

PyTorch to StreamIt Graph Compilation

The compiler developed for converting PyTorch models into StreamIt `stream` serves as an intermediary tool designed to translate the high-level representations of neural network architectures in PyTorch into a format conducive to stream processing applications. The primary function of this compiler is to create a structured, data-flow oriented StreamIt `stream` from a PyTorch computational graph. This process involves several critical steps:

- **Graph Extraction:** The compiler begins by extracting the computational graph from the PyTorch model. This graph outlines the operations (nodes) and their dependencies (edges), representing the flow of tensors through the model's layers and functions. This is done by symbolic execution of the given model. Notice how this is very similar to BuildIt that we discussed earlier, that a library embedded in a mainstream language does symbolic execution and generates a computation graph that can be used for optimizations and generating a next stage code.

- **Intermediate Representation:** There is often a variable reuse in the Python code for PyTorch modules. Also, multiple operations might be nested into one another. To simplify all such situations, I designed an intermediate representation (IR) that has two forms - one is a graph that has edges depicting dependencies but only containing information I care about for further StreamIt `stream` generation and other is a flattened

set of instructions. Each instruction creates a new variable as the output, and is a result of some operation on variables created by preceding instructions.

- **Optimization:** The compiler then does a liveness analysis pass over the IR graph and eliminates dead code. The IR graph lends itself for other such passes as well.

- **Code Generation:** Finally, the compiler generates the StreamIt `stream` code that embodies the translated and optimized model. This code is ready to be utilized in stream processing environments where the high throughput and parallel processing capabilities of StreamIt can be leveraged.

This compiler facilitates the deployment of sophisticated machine learning models via StreamIt to edge devices, heterogeneous and distributed systems.

4.1 Graph Extraction

The `torch.fx` module in PyTorch provides tools for capturing and transforming the computational graphs of PyTorch models, facilitating programmatic manipulation of the model's operations.

4.1.1 Example PyTorch Module

Consider the following PyTorch module:

```
import torch
import torch.nn as nn

class MyModule(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(4, 5)
    def forward(self, x):
        x = self.linear(x)
        return x + torch.sin(x)
```

Figure 4.1: Python code for a simple PyTorch module.

This module introduces a parameter and a linear transformation, followed by a non-linear activation function (sine).

4.1.2 Graph Generation using `torch.fx`

To generate the computational graph of this model, we use `torch.fx` as follows:

```
import torch.fx as fx

model = MyModule()
x = torch.randn(3, 4)  # Example input tensor
graph = fx.symbolic_trace(model)

print(graph)
```

Figure 4.2: Using `torch.fx` to generate the computational graph.

Notice that this is symbolic execution, hence the variable `x` declared for input is not used anywhere.

4.1.3 Graph Output and Explanation

The output from the `print(graph)` looks like this:

```
graph():
    %x : [num_users=1] = placeholder[target=x]
    %linear : [num_users=2] = call_module[target=linear](args = (%x,), kwargs
    = {})
    %sin : [num_users=1] = call_function[target=torch.sin](args = (%linear,),
    kwargs = {})
    %add : [num_users=1] = call_function[target=operator.add](args = (%linear,
    %sin), kwargs = {})
    return add
```

Figure 4.3: Output of the `torch.fx` graph trace.

The table below presents this graph in a structured format:

Opcode	Name	Target	Arguments	Keyword Arguments
`placeholder`	x	x	()	{}
`call_module`	linear	linear	(x,)	{}
`call_function`	sin	<built-in method sin..>	(linear,)	{}
`call_function`	add	<built-in function add>	(linear, sin)	{}
`output`	output	output	(add,)	{}

Table 4.1: Table representation of the `torch.fx` graph.

Here is a visual I've created to see the flow of the tensors.

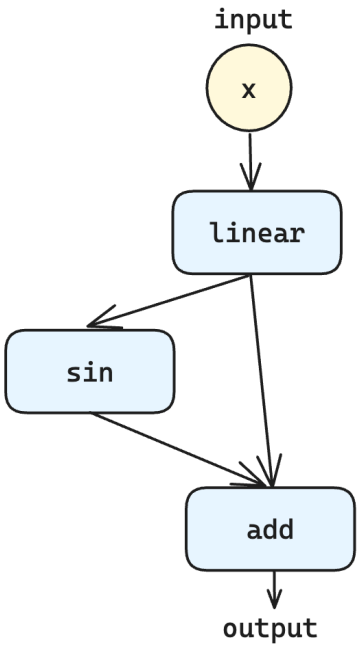

Figure 4.4: Visualizing the computation graph of `MyModule` from 4.2

4.1.4 Explanation of `torch.fx` Features

- **Graph Nodes**: Each operation in the model is represented as a node in the graph. This includes both primitive operations like addition and more complex functions like those provided by PyTorch modules. There are multiple types of nodes.
  - `placeholder` denotes that space for a tensor is allocated. This naturally does not have any arguments but has a name.

- `get_attr` is for finding a parameter from a class, or finding some element from a tuple of outputs.
- `call_function` is for function calls like adding, activation functions etc. These are functions that do not have trainable weights. The built-in function `getitem` is for operations like accessing a slice/view/value from a tensor.
- `call_module` is for calling the forward function of modules (typically neural network layers, but precisely instances of `torch.nn.Module`s.) These differ from functions by having trainable weights.
- `output` denotes what values are returned. This information is very helpful in doing a liveness analyis pass and removing the nodes that do not contribute towards outputs.

Each node also has a name to make it unique.

- **Edges and Data Flow**: The arguments to each node illustrate the flow of tensors through the model. For instance, `add1` takes `x` and `param1` as inputs and `add1` itself is fed into `linear1`.

4.2 Intermediate Representation

The Intermediate Representation (IR) for PyTorch models is designed to simplify the structure of neural network models by transforming complex Python code and nested operations into a more analyzable format. This is achieved through a dual-representation approach:

- **Graph Representation:** This component visualizes the model as a graph with nodes representing operations or data entities and directed edges depicting data dependencies. It focuses on elements crucial for further transformations, particularly for generating StreamIt code.

- **Flattened Instruction Set:** This linear sequence of instructions ensures each operation in the model generates a new variable, avoiding variable reuse and simplifying the computational model for sequential processing and hardware translation.

4.2.1 Syntax of the IR

The IR syntax is designed for clarity and straightforward interpretation, where each instruction specifies an operation and its dependencies explicitly. The formal grammar of the IR is as follows:

```
IR ::= Instruction*
Instruction ::= Var '=' Op [ Params ] '(' VarList ')'
Var ::= 'var' Int
Op ::= [a-zA-Z_] [a-zA-Z0-9_]*
Params ::= '<' ParamList '>'
ParamList ::= Param (',' Param)*
Param ::= Int | Slice
Slice ::= 'slice' '(' SliceArg ',' SliceArg ',' SliceArg ')'
SliceArg ::= IntLit | 'None'
VarList ::= Var (',' Var)*
IntLit ::= [0-9]+
```

`Var` represents a variable, uniquely identified by an integer (identifier) i.e. a variable `var4` has an identifier 4. `Op` denotes an operation applied to one or more variables. This can be a layer with parameters or some built-in function or a slice etc. Each line in the IR describes a unique operation, demonstrating the transformation from input to output.

4.2.2 Mathematical Justification

The PyTorch computation graph will always have an acyclic dependence, hence the computation graph and the IR graph is a directed acyclic graph (DAG.) Having a cyclic dependence is not possible as a value cannot be created from other values that depend on this value itself. The flattened IR code is one of the possibly several topological sortings over the IR graph. This is also ensured by having instructions such that if an instruction represents a function $f : \mathbf{R}^n \to \mathbf{R}$, $v_i = f(v_{i_1}, v_{i_2}, ..., v_{i_n}; params)$, where $v_i, v_{i_1}, .., v_{i_n}$ are variables with identifiers $i, i_1, ..., i_n$, then $i > i_j \forall 1 \leq j \leq n$. Also as a definition, I will refer to v_i as the **result** of the instruction and $v_{i_1}, ..., v_{i_n}$ to be the **arguments** of the instruction.

4.2.3 Example IR

Consider the module `MyModule` in 4.1, the IR for that module will be the following.

```
var0 = placeholder()
var1 = linear<4, 5>(var0)
var2 = sin(var1)
var3 = add(var1, var2)
```

Notice how the variable identifiers are all unique and increase from 0 to 3. Each instruction generates a new variable and the functions depend on variables with lesser identifiers to generate that variable. Also notice how the parameters of the linear layer which are the number of input and output channels have been parsed from the PyTorch graph and encoded in the IR for future use.

4.3 StreamIt Code Generation

Now that the IR is ready, we traverse over the IR graph to generate the equivalent StreamIt graph. The overall algorithm has a lot of details, so I will go over some examples and progressively add complexity.

4.3.1 Simple sequential models

A large proportion of models will be just a series of transformations of the input like this one below:

```
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.conv_layer = nn.Conv2d(in_channels=3, out_channels=32,
    kernel_size=3)
        self.max_pool = nn.MaxPool2d(kernel_size = 2, stride = 2)
        self.flatten = nn.Flatten()
        self.fc = nn.Linear(1600, num_classes)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.conv_layer(x)
        out = self.max_pool(out)
        out = self.flatten(out)
        out = self.fc(out)
        out = self.relu(out)
        return out
```

Figure 4.5: An example of a module that sequentially does operations on the input

Here is what the IR would look like

```
var0 = placeholder()

var1 = conv2d<3, 32, 3, 3, 1, 1, 0, 0>(var0)

var2 = maxpool2d<2, 2, 0>(var1)

var3 = flatten(var2)

var4 = linear<1600, 10>(var3)

var5 = relu(var4)
```

In this example, it is clear that we just need a pipeline in StreamIt. For this, the algorithm by default just creates a pipeline as the top level stream to be returned. Then it iterates over all the IR instructions in the flattened topologically sorted order. Each instruction is analyzed for push, pop and peek rates and added as a filter in the pipeline. Since it is not possible to know all this automatically, we generate a StreamIt code as follows:

```
pipeline<tensor_t, tensor_t>([&](auto pipe) {
  pipe->add(conv2d(3, 32, 3, 3, 1, 1, 0, 0));
  pipe->add(maxpool2d(2, 2, 0));
  pipe->add(flatten());
  pipe->add(linear(1600, 10));
  pipe->add(relu());
});
```

Figure 4.6: Generated StreamIt graph for the CNN module from 4.5

The above code creates a pipeline which has input and output types both as `tensor_t`, which is a mock type for the BuildIt system to generate the `libtorch`'s tensor type in the second stage of code generation. During each call to `pipe->add()`, we are adding a filter. The individual functions are implemented separately, one each for each possible layer, function or operator. It generates a new filter for the given set of parameters.

4.3.2 ResNet - An example for Duplicate Split Join

Now, let us handle split joins. Consider Residual Networks (ResNet) [17] which introduced the idea of having residual edges in deep convolutional neural networks. This means that the input will go through a sequence of convolutional blocks and at the end, the input itself gets added to the result to have a better influence on the result. 4.7 is an example code for ResNet block. Multiple such blocks are typically connected in series, but we will look at this small block for simplicity. Notice in the code that we need a `DuplicateSplitter` as the same value `x` gets passed through the convolutions and also downsample. These two paths join at the addition.

```python
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride = 1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Sequential(
                        nn.Conv2d(in_channels, out_channels, 3, stride, 1),
                        nn.BatchNorm2d(out_channels),
                        nn.ReLU())
        self.conv2 = nn.Sequential(
                        nn.Conv2d(out_channels, out_channels, 3, 1, 1),
                        nn.BatchNorm2d(out_channels))
        self.downsample = nn.Sequential(
                        nn.Conv2d(3, 64, 1),
                        nn.BatchNorm2d(64))
        self.relu = nn.ReLU()
        self.out_channels = out_channels

    def forward(self, x):
        out = self.conv1(x)
        out = self.conv2(out)
        residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out
```

Figure 4.7: ResNet Code - An example of DuplicateSplitJoin

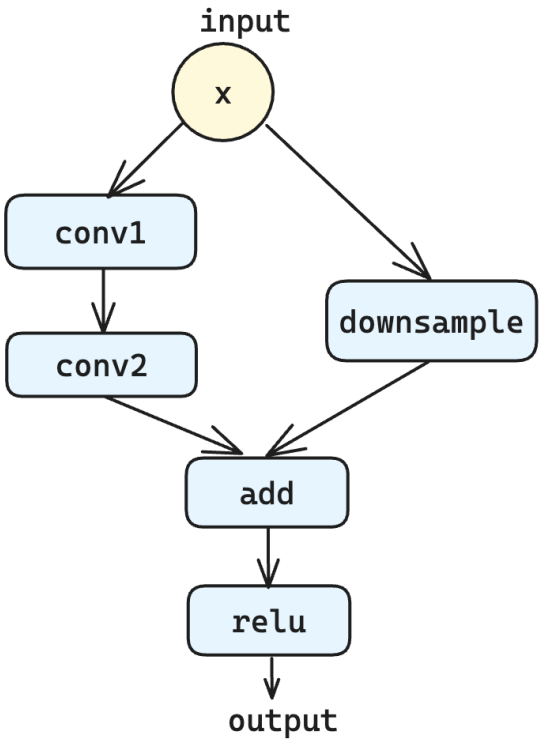

Figure 4.8: PyTorch Graph visualization for 4.7

To handle this, I revised the iteration scheme from just plainly iterating over a flattened IR instruction list to doing depth first search (DFS) traversal over the IR graph. At each vertex (instruction), if the result of that instruction is fed as it is to more than one instructions, then we create a duplicate split join. So, to the top level pipeline, we add a split join, and we keep the split join open. When we travel on each outgoing edge, we create a new stream inside the split join. The new streams are created recursively via another recursive DFS traversal. Note that we have not yet closed the split join with a joiner. Hence, I do not traverse a depth fully. When I reach an instruction that has multiple arguments, I know that that is a join. Hence, I return back to let other streams in the split join reach the join. When the last stream has reached the joining instruction, I close the split join with a round robin joiner, and put the joining instruction as the next filter in the top-level pipeline. Notice that each individual stream in the split join will be a top-level pipeline in itself while it is getting generated and it can have split joins within it too. Also, notice that I put the joining instruction outside of the split join. This is because a join is created in a PyTorch graph when a function or operation requires multiple arguments created as results in previous instructions. This means that its pop rate and peek rate is more than one. So, if we interleave the output channels of the split join streams with a round robin joiner, we can pop multiple data points at once. For example, in the ResNet graph, add requires 2 tensors. Each call to the work function will generate one output from the conv1 - conv2 stream and one output from downsample. The round robin joiner takes one from each and push two outputs into the stream forward. These two are absorbed into add. Here is the StreamIt code generated for `ResNet(in_channels = 3, out_channels = 64)` and after that a visualization of the StreamIt graph.

```
pipeline<tensor_t, tensor_t>([&](auto pipe) {
  pipe->add(duplicateSplitJoin<tensor_t, tensor_t>([&](auto sj) {
    sj->add(1, pipeline<tensor_t, tensor_t>([&](auto pipe) {
      pipe->add(conv2d(3, 64, 3, 3, 1, 1, 1, 1));
      pipe->add(batchnorm2d(64));
      pipe->add(relu());
      pipe->add(conv2d(64, 64, 3, 3, 1, 1, 1, 1));
      pipe->add(batchnorm2d(64));
    }));
    sj->add(1, pipeline<tensor_t, tensor_t>([&](auto pipe) {
      pipe->add(conv2d(3, 64, 1, 1, 1, 1, 0, 0));
      pipe->add(batchnorm2d(64));
    }));
  }));
  pipe->add(add());
  pipe->add(relu());
});
```

Figure 4.9: Generated StreamIt code for ResNet

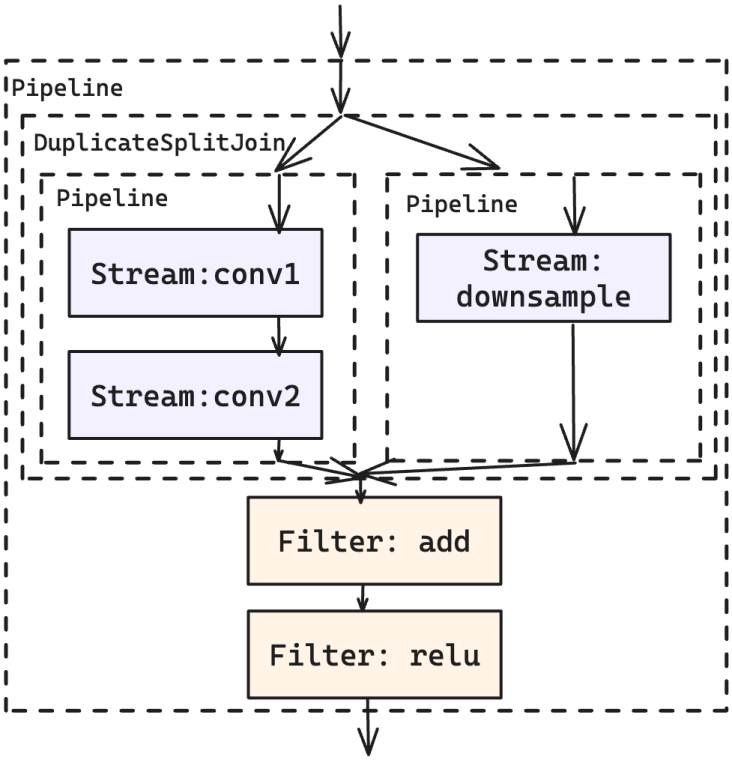

Figure 4.10: Visualization of StreamIt stream for ResNet

Notice that I have not expanded on the streams created for conv1, conv2 and downsample. These are instances of `nn.Sequential` which creates a pipeline for each of them.

4.3.3 Round Robin Split Joins

Our examples so far have only one input and one output. However, this may not always be the case. Modules can accept more than one input and more than one output. However, our current scheme of DFS traversal will just try to close a split join, if and when the multiple inputs ever meet at one instruction. To correct for that, I introduce a root node that is connected to all the instructions that did not have an incoming edge. However, notice here that we cannot create a duplicate split join, as we want a round robin split join with weights 1 each so that each input goes to the stream that processes each input. Similarly at the output, I add an additional node that connects all the instructions that have no outgoing edge to create a single interleaved stream, maintaining the StreamIt abstraction. Another place where we can get round robin split joins is if a particular function returns multiple outputs in a tuple, and each output is handled separately afterwards. Now, to determine if an instruction having multiple outgoing edges needs to be a duplicate split join or a round robin split join, I created a check. If a split occurs at an instruction, then it will only happen when the exact result of that instruction is the argument of multiple future instructions. Now, if the downstream filters are `getitem` tensors, which means they are accessing different parts of the input, then I create a round-robin split join, else a duplicate split join. On the other hand, if this split is at the root, then it is always a round-robin split join. Here are a few examples for this.

```python
class TwoInputs(nn.Module):
    def __init__(self):
        super(TwoInputs, self).__init__()
        self.conv1 = nn.Sequential(nn.Conv2d(3, 64, 3), nn.BatchNorm2d(64))
        self.conv2 = nn.Conv2d(3, 64, 3)

    def forward(self, x, y):
        return self.conv1(x) + self.conv2(y)
```

Figure 4.11: Two Inputs

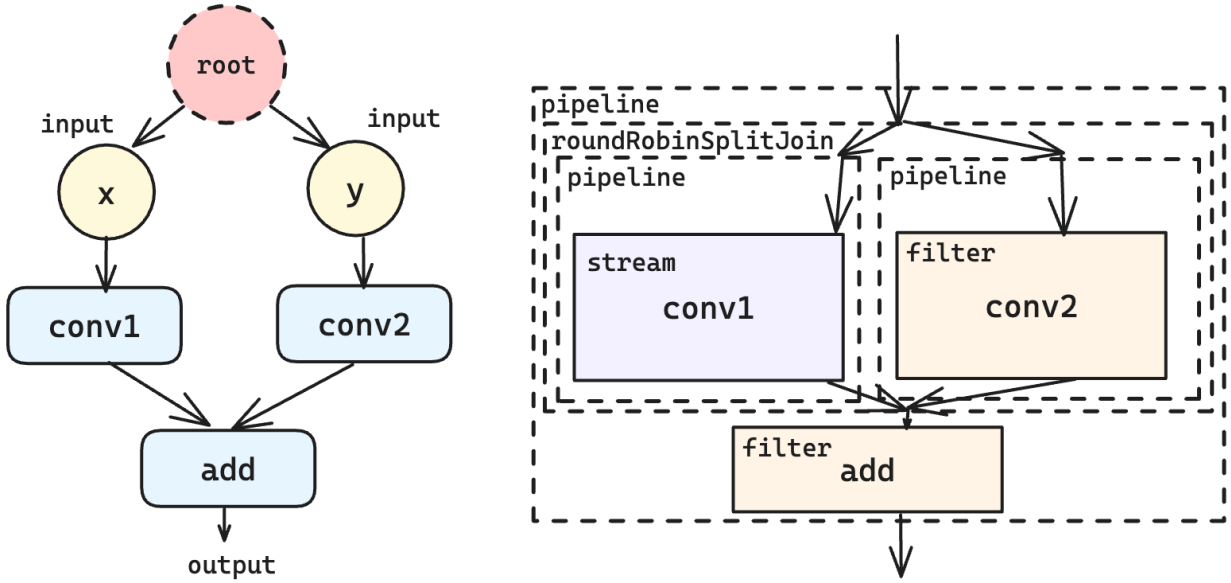

Figure 4.12: PyTorch graph (left) and StreamIt stream visualization (right) for Two Inputs

```
pipeline<tensor_t, tensor_t>([&](auto pipe) {
  pipe->add(roundRobinSplitJoin<tensor_t, tensor_t>([&](auto sj) {
    sj->add(1, 1, pipeline<tensor_t, tensor_t>([&](auto pipe) {
      pipe->add(conv2d(3, 64, 3, 3, 1, 1, 0, 0));
      pipe->add(batchnorm2d(64));
    }));
    sj->add(1, 1, pipeline<tensor_t, tensor_t>([&](auto pipe) {
      pipe->add(conv2d(3, 64, 3, 3, 1, 1, 0, 0));
    }));
  }));
  pipe->add(add());
});
```

Figure 4.13: Two Inputs StreamIt code

```python
class TwoOutputs(nn.Module):
    def __init__(self):
        super(TwoOutputs, self).__init__()
        self.conv1 = nn.Sequential(nn.Conv2d(3, 64, 3), nn.BatchNorm2d(64))
        self.conv2 = nn.Conv2d(3, 64, 3)

    def forward(self, x):
        return self.conv1(x), self.conv2(x)
```

Figure 4.14: Two Outputs

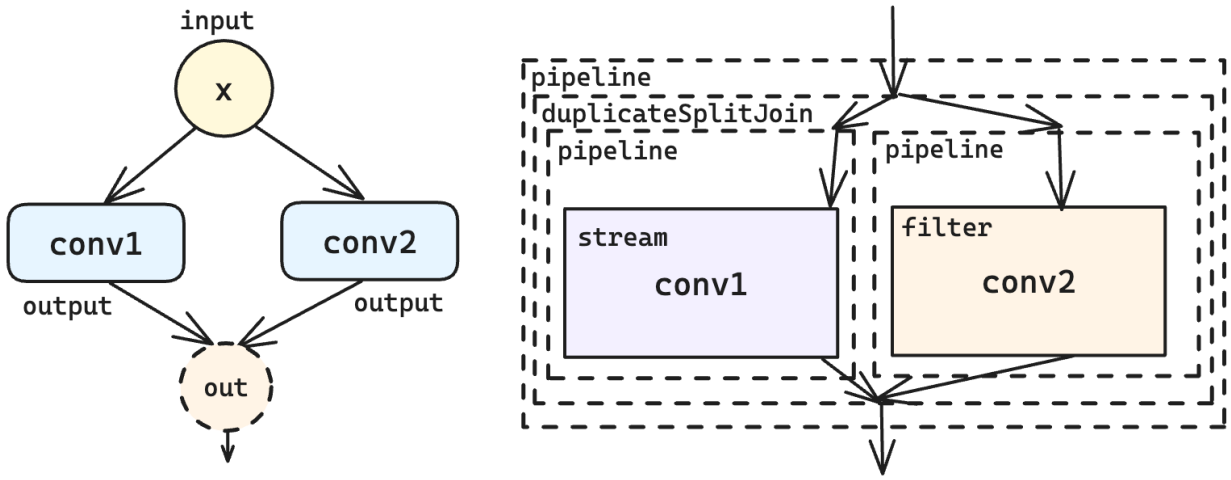

Figure 4.15: PyTorch graph (left) and StreamIt stream visualization (right) for Two Outputs

```
pipeline<tensor_t, tensor_t>([&](auto pipe) {
  pipe->add(duplicateSplitJoin<tensor_t, tensor_t>([&](auto sj) {
    sj->add(1, pipeline<tensor_t, tensor_t>([&](auto pipe) {
      pipe->add(conv2d(3, 64, 3, 3, 1, 1, 0, 0));
      pipe->add(batchnorm2d(64));
    }));
    sj->add(1, pipeline<tensor_t, tensor_t>([&](auto pipe) {
      pipe->add(conv2d(3, 64, 3, 3, 1, 1, 0, 0));
    }));
  }));
});
```

Figure 4.16: Two Outputs StreamIt code

4.3.4 Liveness Analysis and Dead Code Elimination

Look at this example PyTorch module.

```
class LSTMModel(nn.Module):
    def __init__(self):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(28, 128, 2, batch_first=True)
    def forward(self, x):
        out, _ = self.lstm(x, None)
        return out
```

Figure 4.17: LSTM example to motivate dead code elimination

Here, the `self.lstm` returns a tuple of two values, but only one is used. The PyTorch graph created by default is as follows. Now, if we follow the algorithm that we have to

Table 4.2: `torch.fx` graph for `LSTMModel`

opcode	name	target	args	kwargs
placeholder	x	x	()	{}
call_module	lstm	lstm	(x, None)	{}
call_function	getitem	<built-in function getitem>	(lstm, 0)	{}
call_function	getitem_1	<built-in function getitem>	(lstm, 1)	{}
output	output	output	(getitem,)	{}

generate a StreamIt stream, then we will generate the following IR.

```
var0 = placeholder()
var1 = lstm<28, 128, 2>(var0)
var2 = getitem<0>(var1)
var3 = getitem<1>(var1)
```

Then, we will detect that the result `var1` has two outgoing edges. We generate a split join. However, we never get to a join because only one output exists. To resolve this issue, we do a liveness analysis. All outputs are marked to be live and then we traverse the graph backwards and mark each instruction visited as live. Then while traversing, I ignore the dead nodes. While determining whether there is need for a split join, I check for the live

children and not all of them. So now we know that we will ignore the instruction `var3 = getitem<1>(var1)` in this. But, the output channel from `lstm` is generating two outputs for every input. We cannot just remove the instruction `var3 = getitem<1>(var1)` and pass all the output data from `lstm` into `var2 = getitem<0>(var1)`. For this, I introduce the concept of a `separator` filter. I add this filter after every such case where one instruction is generating more than one output results but not all of them are being accepted into live streams. The separator filter takes in a vector of weights and denotes how many repetitions of each output from an instruction are needed downstream. If an output is dead, then that has a weight 0.

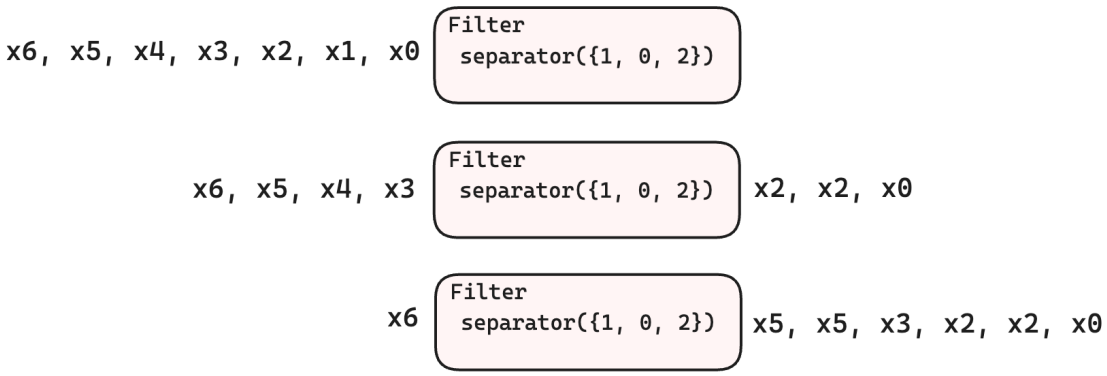

Figure 4.18: Separator Filter in Action

Below I show the transformation of the PyTorch graph and the resulting StreamIt code and graph.

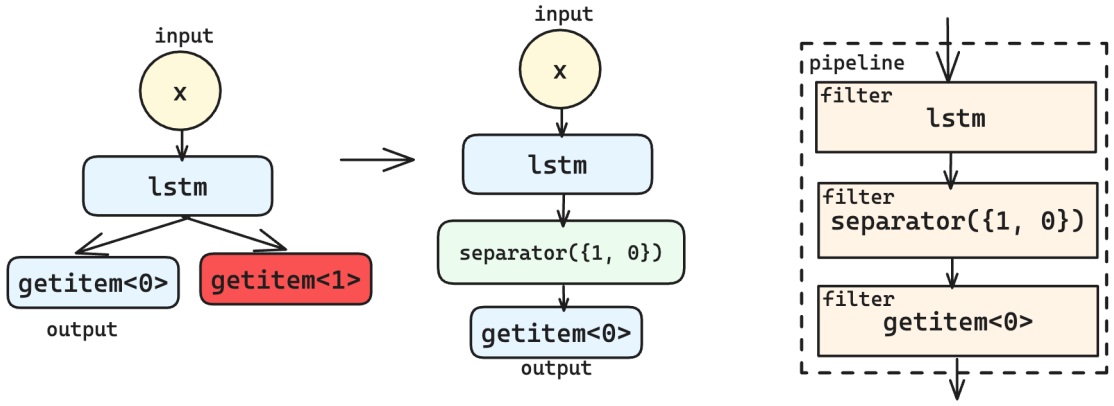

Figure 4.19: Dead Code Elimination on the IR graph (left), StreamIt stream (right)

```
pipeline<tensor_t, tensor_t>([&](auto pipe) {
  pipe->add(lstm(28, 128, 2));
  pipe->add(separator({1, 0}));
  pipe->add(getitem(0));
});
```

Figure 4.20: Output StreamIt code for `LSTMModel`

Notice how there is not split join also in the final output code.

4.3.5 Fused Instructions

There are a few fairly common programming patterns that will require some modifications to the StreamIt abstraction or some performance loss. Consider the following module

```
class Fused(nn.Module):
    def __init__(self):
        super(Fused, self).__init__()
        self.f = nn.Linear(4, 5)
        self.g = nn.Linear(4, 1)
        self.h = nn.ReLU()
    def forward(self, x):
        y = self.g(x)
        return y + self.f(x), y + self.h(x)
```

Figure 4.21: Fused Split Join example. f, g, h are some functions

Table 4.3: `torch.fx` graph for `Fused`

Opcode	Name	Target	args	kwargs
placeholder	x	x	()	{}
call_module	g	g	(x,)	{}
call_module	f	f	(x,)	{}
call_function	add	<built-in function add>	(g, f)	{}
call_module	h	h	(x,)	{}
call_function	add_1	<built-in function add>	(g, h)	{}
output	output	output	((add, add_1),)	{}

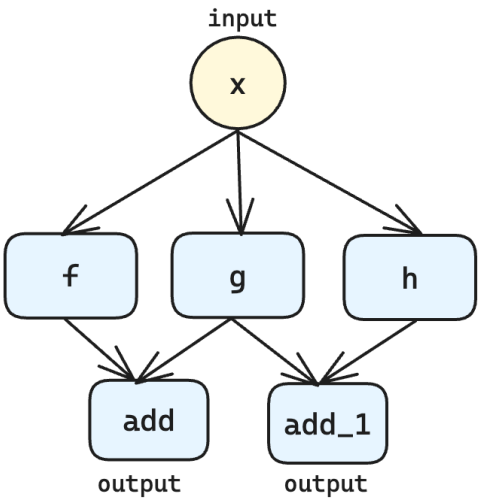

Figure 4.22: Visualizing PyTorch graph for `Fused`

For this case, notice that in the current state this graph cannot be converted to a StreamIt graph as there is no clean split join. The two split joins have a common node at g. There are 2 ways to resolve this -

- Detect such cases by traversing the graph backwards from each join. Create a copy of each instruction seen this way and recreate the graph. Eventually, we will have a separate split join for each join instruction. At the inputs, if there are multiple copies of the input, replace it with a single node and a duplicate splitter. I have currently implemented a **fused instruction replication** pass that does exactly this. However, this means that the fused instructions are executed multiple times which is wasteful.

- Relax the StreamIt abstraction from SplitJoins to splitters and joiners. This however would require a much higher effort on the side of StreamIt compiler. Also, this might break some optimizations that rely on the StreamIt abstraction. And lastly, this essentially makes StreamIt the same as connecting individual nodes in a graph losing the appeal of StreamIt.

4.3.6 Sequence Models - Static vs Dynamic Loops, Back Edges

Sequence models such as Recurrent Neural Networks (RNNs), Long Short Term Memory (LSTMs) and Gated Recurrent Units (GRUs) work in a similar way. They have a single set of weights for all tokens in a sequence. The tokens appear sequentially, for each token, the operations are done and an output and a hidden state are generated. This hidden state and the next token are the inputs for the next time step, the function and the weights remaining the same. Hence it is a very popular technique to create a module for just one time step. This block contains the weights and one call to the forward function takes in a hidden state and a token and returns output and next hidden state. Then another module does this for as many tokens are there. For this, we need a for loop. As we discussed in Chapter 3, a key advantage of PyTorch dynamic graphs is the ability to only call the recurrent function on tokens that exist in the sequence and sequences could be of variable lengths. However such a code is not possible to symbolically execute because the number of loop iterations for the loop that iterates over the sequence and applies the recurrent functions on it, is dynamic i.e., dependent on the input. If this size is static i.e. known before runtime, then the loop gets unrolled just like BuildIt loops get unrolled when iterated by a static variable. This means we have to preselect the maximum possible sequence length as a static value and unroll the loop. This approach also solves the issue of a back edge because without unrolling, the hidden state represented a back edge which is not possible to encode in the version of StreamIt we presented in Chapter 2.

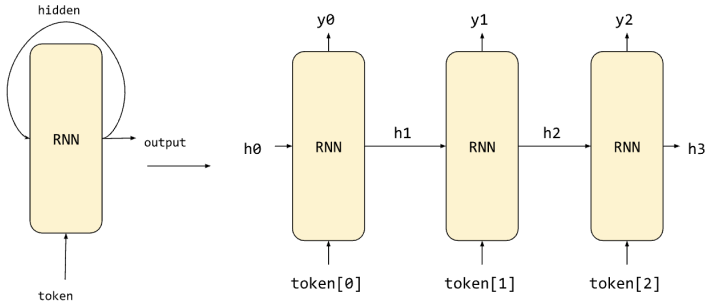

Figure 4.23: Dynamic vs Unrolled RNN

Chapter 5

Experiments & Conclusion

In order to split a StreamIt graph of a machine learning model on a multicore system, we will have to split on some dimension of the input tensor. The key experiments run to evaluate the performance of this project are trying to split tensors on various dimensions are running this code on multiple number of cores and analyzing the performance to see if splitting tensors into multiple parts is a good idea or not.

I used OpenCilk to parallelize the split tensor operations and compared the performance with libtorch.

I measure the performance of a linear layer computation, split on the batch size. I split it into two parts and then, in general, with the granularity as a parameter. The performance with split tensors is not optimal without some optimizations on not copying the tensors, which will be developed by the group in the future. However, we can see that when tensor is split into a granularity significantly higher than 1, the performance is comparable (within 3-4x) to the performance of libtorch. We believe that this is entirely attributed to the copying of the results of each split tensor into the result, and also due to some vectorization that breaks with our splitting.

Number of Workers	libtorch	Split in 2	granularity 64	granularity 128
1	253284ns	921548ns	1075967ns	859909ns
2	240746ns	922122ns	862070ns	745538ns
4	242876ns	898322ns	654500ns	626017ns
8	249802ns	899796ns	716605ns	697393ns

Table 5.1: Linear layer: 1024x512 inputs, 1024x256 outputs with split tensors

Number of Workers	libtorch	Split in 2	granularity 64	granularity 128
1	6565μs	18718μs	22376μs	22455μs
2	6384μs	19746μs	24404μs	22796μs
4	6359μs	19143μs	23698μs	20490μs
8	6490μs	19150μs	23546μs	20854μs

Table 5.2: Linear layer: 1024x4096 inputs, 1024x1024 outputs with split tensors

5.1 Future Work

- Developing a scheduler to parallelize StreamIt graphs
- A system that views StreamIt edges as network communication and individual nodes as being run on different machines in order to run these models on distributed systems.
- Modifying torch tensor implementation in such a way that when we split workloads on a single machine, no copying of tensors is required.

5.2 Conclusion

From the work of this thesis I conclude that multistage code generation is a fast and powerful way to compile StreamIt graphs, thus improving the StreamIt compiler. Also, StreamIt provides a useful abstraction for splitting and distributing machine learning workloads, hence compiling PyTorch models into StreamIt graphs is a very crucial contribution.

References

[1] W. Thies, M. Karczmarek, M. I. Gordon, D. Maze, J. Wong, H. Hoffman, M. Brown, and S. Amarasinghe, "Streamit: A compiler for streaming applications," *MIT/LCS Technical Memo LCS-TM-622*, Dec. 2001.

[2] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," *MIT/LCS Technical Memo LCS-TM-620*, Aug. 2001.

[3] A. Brahmakshatriya and S. Amarasinghe, "Buildit: A type based multistage programming framework for code generation in c++," in *Proceedings of 2021 International Symposium on Code Generation and Optimization*, 2021.

[4] "The llvm compiler infrastructure." (), URL: https://github.com/llvm/llvm-project/ (visited on 05/14/2024).

[5] A. Brahmakshatriya and S. Amarasinghe, "Graphit to cuda compiler in 2021 loc: A case for high-performance dsl implementation via staging with buildsl," in *Proceedings of 2022 International Symposium on Code Generation and Optimization*, 2022.

[6] A. Brahmakshatriya and S. Amarasinghe, "D2x: An extensible contextual debugger for modern dsls," in *Proceedings of 2023 International Symposium on Code Generation and Optimization*, 2023.

[7] "Pytorch." (), URL: https://pytorch.org/ (visited on 05/14/2024).

[8] "Tensorflow." (), URL: https://www.tensorflow.org/ (visited on 05/14/2024).

[9] "Introduction to graphs and tf.function." (), URL: https://www.tensorflow.org/guide/intro_to_graphs (visited on 05/15/2024).

[10] "Overview of pytorch autograd engine." (Jun. 8, 2021), URL: https://pytorch.org/blog/overview-of-pytorch-autograd-engine/ (visited on 05/15/2024).

[11] "Tensorflow graph optimization with grappler." (), URL: https://www.tensorflow.org/guide/graph_optimization (visited on 05/15/2024).

[12] R. Socher. "Cs224d: Deep learning for natural language processing," Stanford University. (2016), URL: https://cs224d.stanford.edu/lectures/CS224d-Lecture10.pdf (visited on 05/15/2024).

[13] R. Yadav, A. Aiken, and F. Kjolstad, "Distal: The distributed tensor algebra compiler," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022, San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 286–300, ISBN: 9781450392655. DOI: 10.1145/3519939.3523437. URL: https://doi.org/10.1145/3519939.3523437.

[14] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 77:1–77:29, Oct. 2017, ISSN: 2475-1421. DOI: 10.1145/3133901. URL: http://doi.acm.org/10.1145/3133901.

[15] S. Li. "Pytorch distributed overview." (), URL: https://pytorch.org/tutorials/beginner/dist_overview.html (visited on 05/15/2024).

[16] S. Li, Y. Zhao, R. Varma, *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, 2020, ISSN: 2150-8097. DOI: 10.14778/3415478.3415530.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.