Converting PyTorch Models to StreamIt Pipelines

by

Muhender Raj Rajvee

Bachelor of Science in Computer Science and Engineering,
Massachusetts Institute of Technology, 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:	Muhender Raj Rajvee Department of Electrical Engineering and Computer Science May 16, 2025
Certified by:	Saman Amarasinghe Professor of Electrical Engineering and Computer Science, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Converting PyTorch Models to StreamIt Pipelines

by

Muhender Raj Rajvee

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

With the rise of large language models, there have been efforts to optimize machine learning inference to support a large volume of queries. Currently, the two main ways to do this are running optimized kernels for computing the forward inference pass and distributing computation across multiple GPUs or different cores in a GPU. Machine learning libraries such as PyTorch produce dynamic computation graphs in order to represent the forward pass of the model. PyTorch allows conversion of these dynamic graphs into static ones through just-in-time (JIT) compilation. These graphs can then be optimized further by the compiler. We propose an alternate way of optimizing these dynamic graphs. We convert the dynamic computation graph of PyTorch to pipelines in StreamIt, a domain specific language (DSL) for streaming applications, and use the multi-stage compilation property of BuildIt to compile this pipeline in stages to inference code. We found that, while the inference latencies of models compiled in this way are slightly higher, they are still comparable to those of PyTorch models and are open to future optimizations.

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

Acknowledgments

I express my deepest gratitude to Saman Amarasinghe for suggesting an interesting topic for my thesis work. His suggestions helped me a lot in shaping the content and direction of my thesis.

I am grateful to doctoral student Ajay Brahmakshatriya, who helped me a lot with the technical parts of my thesis. He suggested several improvements to my code and helped fix technical issues. I also appreciate his technical insights about project structure and code quality.

I am also thankful to Kaustubh Dighe, a previous MEng student in the lab. His thesis work served as a starting point for this work, and I borrowed and expanded on a lot of ideas from it.

I am thankful to my parents for their constant support in my education. Their encouragement and insistence on the highest standards for my work early on in life are some of the main contributors to my achievements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As machine learning model sizes get bigger and Moore's law comes to an end, there is an increasing need to split computation between several GPUs in a distributed fashion in order to maintain low latencies. This can be done in several different ways, one such way being pipeline parallelism. In pipeline parallelism, each GPU holds one layer of the model and performs the corresponding computations in a pipelined manner, improving the overall inference throughput. This improvement is especially noticeable in the case of large language model (LLM) inference.

1.1 Related Work

We discuss a few ways in which high-performance machine-learning applications are currently built.

1.1.1 PyTorch Tracing vs Scripting

PyTorch is a Python library that provides GPU-accelerated tensor computation and the ability to use these operations for building deep neural networks (DNNs) [1]. PyTorch can execute a trace of a model to determine the sequence of operations performed on an input.

This trace could then be exported and used in other non-Python contexts. However, the major disadvantage is that tracing only follows one possible control flow path and eliminates all control flows by choosing a particular branch.

PyTorch scripting solves this problem by statically analyzing the model and all its branches, producing a TorchScript model that captures the control flow behavior of the model on various attributes. However, having control flow as part of the model can make it slow, so tracing and scripting can be used together to fine-tune a model's definition.

1.1.2 PyTorch compile

PyTorch has a built-in just-in-time (JIT) compiler that can be used to generate optimized kernels on the fly. Previously, the only way to access this compiler was through tracing and scripting as described above. The new torch.compile function introduced in PyTorch 2.0 can now be used to JIT compile the model where necessary to speed up execution with minimal code changes. Generally, first-time execution times can be longer than simply interpreting the Python code, even though it pays off eventually after subsequent inferences. This is because JIT compilation into optimal kernels takes some time the first time it is run, leading to cold-start delays.

1.1.3 Other Optimization Methods

There are a number of other methods people currently use to optimize inference. Intel's open source platform, Open Visual Inference and Neural Network Optimization (OpenVINO), is commonly used to deploy optimized inference models [5]. OpenVINO uses techiques such as fusion of multiple layers and removing unnecessary layers to speed up inference.

1.2 Using StreamIt for Machine Learning

This thesis proposes using StreamIt to enable more static graph optimizations in an easier manner. StreamIt is a language originally intended for signal processing applications [6]. However, the structure of StreamIt graphs makes it well suited for pipeline parallelism of machine learning models, where streams of tensors flow between various compute nodes in the graph. Each node can be either a Filter or a SplitJoin. A filter converts a stream from one channel to another, while a SplitJoin distributes streams between multiple nodes and joins them back into a single node. Each StreamIt filter has a `work()` function responsible for performing the node's computation. A SplitJoin can be useful in splitting work between different GPU cores for parallel processing or in distributing work across multiple GPUs over a network.

1.2.1 PyTorch To StreamIt

A previous project with the group aimed to compile PyTorch models to StreamIt graphs for inference [3]. The project made use of PyTorch FX's `symbolic_trace` method to capture the function call graph of the PyTorch model. This graph is then converted to a StreamIt pipeline graph preserving its inference behavior. This resulting StreamIt code can then be compiled with StreamIt libraries and the resulting code could be linked with LibTorch. Since in inference the model's weights are already known, they can be expanded at compile time and all conditions based on the weights can be pre-evaluated.

1.3 Contributions

The objective of this work is to make end-to-end machine learning inference possible via StreamIt graphs. Prior work used in this process is elaborated on in chapter 2. My contribution to the field is in two parts:

1. Linking a StreamIt machine learning pipeline with the actual LibTorch API functions to be able compile it into an executable. This involves adding new types and constructs to StreamIt to be able to describe machine learning layers. This also includes adding functionality for loading the model weights onto certain filters in the StreamIt pipeline (chapter 3).

2. Building on prior work of converting a PyTorch model to a StreamIt pipeline definition. This includes auto-generating StreamIt filter and type definitions for different layer types while still allowing for manual overriding through a configuration file (chapter 4).

Chapter 2

Background and Prior Work

2.1 PyTorch and LibTorch

PyTorch API functions are wrappers of LibTorch kernel code written in C++. Depending on certain properties like tensor sparsity, use of GPUs, and the datatype, PyTorch dynamically dispatches to one of multiple kernels optimized for that set of inputs.

PyTorch is built around the ATen tensor library augmented with Autograd to allow automatic differentiation required for ML training. This involves recording gradients for every operation during the forward pass and traversing the call graph backwards to update them during the backward pass. The backward pass is not required during inference as the weights are not being updated, so users can turn off automatic differentiation when it is not needed to avoid unnecessary overhead.

PyTorch also provides constructs for specifying various distributed paradigms. Users can also choose the collective communications backend that suits their needs. The currently supported backends are Microsoft's Message Passing Interface (MPI), Gloo, and the Nvidia Collective Communications Library (NCCL).

While PyTorch's structure works for most workloads, ones that demand low latencies are

[1]Credit: https://pytorch.org/blog/overview-of-pytorch-autograd-engine/

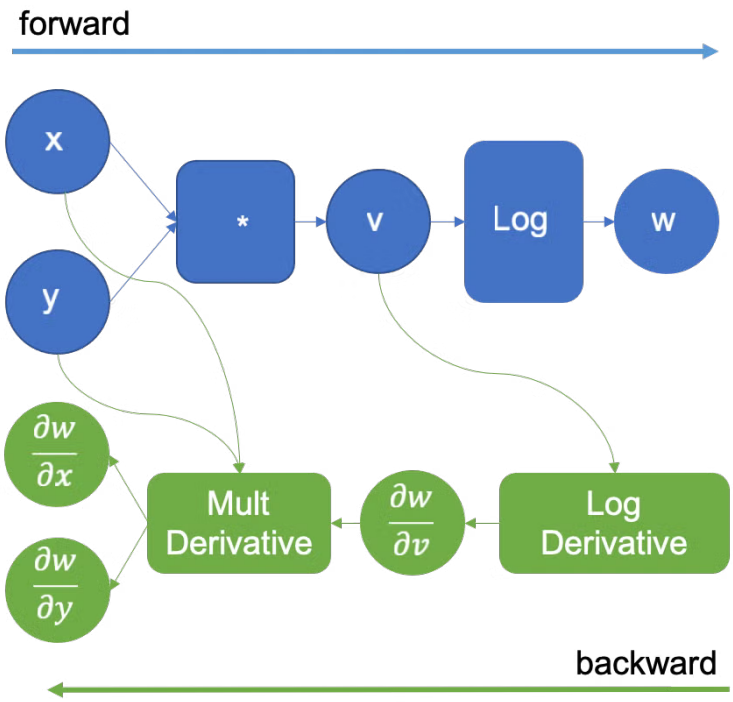

Figure 2.1: PyTorch forward and backward graphs[1]

slowed down by Python's interpreted nature. While individual kernels are highly optimized, there is still room for further optimizations such as operator fusion, where multiple operations are optimized together. This project explores alternative ways of optimizing machine learning algorithms by using an intermediate representation in StreamIt, a language for streaming applications, that is conducive to further optimizations and easy distributed scheduling.

2.2 BuildIt

BuildIt is a framework that allows for easy specification of multi-stage compilation and is used to develop optimized compilers for domain specific languages (DSLs) [2]. BuildIt eliminates complex hierarchies like classes, virtual functions, and inheritance, and produces C code as a flattened version without the overhead. This makes it well-suited for optimizing library calls which often involve several layers of intermediate function calls to choose the right kernel code to execute. Since the StreamIt compiler is written using BuildIt, we can

use BuildIt constructs to generate more optimal code.

2.3 StreamIt

StreamIt uses the multi-stage compilation features of BuildIt to generate optimized code from pipelines. We will use the StreamIt graph in Figure 2.2 to discuss the various StreamIt features that can be used to represent machine learning computation.

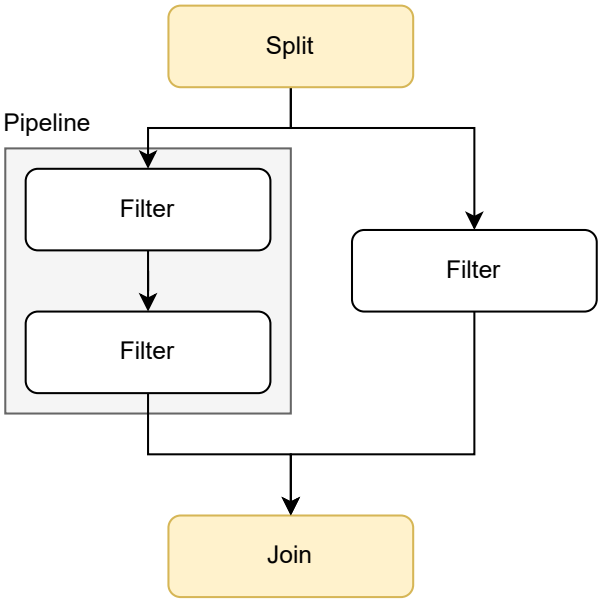

Figure 2.2: An example StreamIt Graph

Pipeline

A pipeline in StreamIt consists of a sequence of other StreamIt constructs like Streams, Filters, or SplitJoins that are logically connected one after the other. Inputs enter the pipeline at the topmost Stream and outputs leave at the bottommost Stream.

This sequential structure of StreamIt pipelines makes it well-suited to represent individual layers in machine learning models, since the activations in these models also follow a similar

pattern.

SplitJoin

There are two main kinds of SplitJoins in StreamIt: Duplicate and RoundRobin. Duplicate SplitJoins create copies of elements in the input channel to feed to each split, while RoundRobin SplitJoins distribute elements from the input channel among the splits in a round-robin manner.

SplitJoins are used whenever there is a split in the computation graph. This could be a single layer's output being passed into two separate layers and the results being combined into a single stream again. This typically happens in residual networks like ResNet which learn the weights of the function F(x)+x rather than just F(x) [4]. This leads to a split graph with the layers computing F(x) on one split and the identity function on the other, joining at a graph node computing the addition of two tensors.

Filter

This represents a transformation of data in the graph. A Filter has a user-specifiable work function associated with it to describe the filter operation. We can thus use Filters in machine learning computation to transform an input channel of tensors into an output channel of tensors.

2.4 Converting PyTorch Graphs to StreamIt Pipelines

The lab has done prior work using the StreamIt framework as an alternative to `torch.compile` for PyTorch graphs [3]. We can use StreamIt constructs such as Filters, Channels, Streams, Pipelines, and SplitJoins to describe an ML model. The computation graph of a PyTorch model is structured as a directed acyclic graph (DAG), which enforces a partial order on the nodes. This structure is very similar to that of a StreamIt pipeline, as discussed above.

2.4.1 The torch.fx API

PyTorch exposes the torch.fx API to trace through a PyTorch model and extract the dynamic layer call graph from it. It extracts this graph by passing Proxy class objects as inputs to the forward function of a model. It then records the operations performed on this object and constructs the dynamic graph. It is possible to extract the generated IR code out of this generated graph.

We will walk through the process of how this prior work extracts the IR from this graph through an example. Figure 2.3 contains PyTorch code for an MNIST model which has the same layers as the StreamIt pipeline we discussed in the previous chapter in Figure 3.13.

```
1
2  class MNIST(nn.Module):
3      def __init__(self):
4          super(MNIST, self).__init__()
5          self.conv_layers = nn.Sequential(
6              nn.Conv2d(1, 32, kernel_size=(3, 4)),
7              nn.ReLU(),
8              nn.Conv2d(32, 64, kernel_size=3),
9              nn.ReLU(),
10             nn.Conv2d(64, 64, kernel_size=3),
11             nn.ReLU()
12         )
13         self.fc_layers = nn.Sequential(
14             nn.Flatten(),
15             nn.Linear(64 * 22 * 22, 10)
16         )
17
18     def forward(self, x):
19         x = self.conv_layers(x)
20         x = self.fc_layers(x)
21         return x
```

Figure 2.3: PyTorch MNIST code

Using the `torch.fx` API to extract the dynamic call graph of this model results in the output in Table 2.1. PyTorch has its own internal representation of the graph which it can use to generate better-optimized code and eliminate unused layers. An example of this IR is given in Figure 2.4.

PyTorch calls each point of computation in this graph a "Node" instance. A 2-dimensional

opcode	name	target	args	kwargs
placeholder	x	x	()	{}
call_module	conv_layers_0	conv_layers.0	(x,)	{}
call_module	conv_layers_1	conv_layers.1	(conv_layers_0,)	{}
call_module	conv_layers_2	conv_layers.2	(conv_layers_1,)	{}
call_module	conv_layers_3	conv_layers.3	(conv_layers_2,)	{}
call_module	conv_layers_4	conv_layers.4	(conv_layers_3,)	{}
call_module	conv_layers_5	conv_layers.5	(conv_layers_4,)	{}
call_module	fc_layers_0	fc_layers.0	(conv_layers_5,)	{}
call_module	fc_layers_1	fc_layers.1	(fc_layers_0,)	{}
output	output	output	(fc_layers_1,)	{}

Table 2.1: MNIST torch.fx symbolic trace tabular output

convolutional layer in the example, say `conv_layers_0` has a node associated with it, with the name consistent with the fully qualified identifier of the layer registered by PyTorch. This node is of type "`call_module`". There are 6 main opcodes of nodes depending on its function.

2.4.2 Node opcodes

1. `call_function`: This node describes a function call. This node is typically used to describe operations like the rectified linear unit (ReLU) activation function, but it also includes PyTorch functions from the `torch.nn.functional` module.

2. `call_module`: This node describes a call to a registered module's forward method. Modules are used to describe the neural network layers that have state. For example, the Linear module holds the linear layer's weights and biases. These modules also compute the backward graph gradients automatically unless explicitly turned off.

3. `call_method`: This node describes a call to a method on the tensor class.

4. `get_attr`: This node describes an attribute access on a Python class. However, it should be noted that PyTorch will not convert explicit calls to the `get_attr` function to this type. These "`call_function`" nodes will then need to be manually converted to this type on parsing the dynamic graph.

```
1
2 graph():
3     %x : [num_users=1] = placeholder[target=x]
4     %conv_layers_0 : [num_users=1] = call_module[target=conv_layers.0](
    args = (%x,), kwargs = {})
5     %conv_layers_1 : [num_users=1] = call_module[target=conv_layers.1](
    args = (%conv_layers_0,), kwargs = {})
6     %conv_layers_2 : [num_users=1] = call_module[target=conv_layers.2](
    args = (%conv_layers_1,), kwargs = {})
7     %conv_layers_3 : [num_users=1] = call_module[target=conv_layers.3](
    args = (%conv_layers_2,), kwargs = {})
8     %conv_layers_4 : [num_users=1] = call_module[target=conv_layers.4](
    args = (%conv_layers_3,), kwargs = {})
9     %conv_layers_5 : [num_users=1] = call_module[target=conv_layers.5](
    args = (%conv_layers_4,), kwargs = {})
10    %fc_layers_0 : [num_users=1] = call_module[target=fc_layers.0](args =
    (%conv_layers_5,), kwargs = {})
11    %fc_layers_1 : [num_users=1] = call_module[target=fc_layers.1](args =
    (%fc_layers_0,), kwargs = {})
12    return fc_layers_1
```

Figure 2.4: PyTorch MNIST generated IR

5. `placeholder`: This is a node representing an argument passed to a layer.

6. `output`: This is a node containing the outputs of the model.

2.4.3 Functions vs Modules

While "`call_function`", "`call_method`", and "`get_attr`" are straightforward, we need to convert "`call_module`" to its equivalent "`call_function`". Almost all PyTorch modules have pure functional equivalents defined in the torch.nn.functional Python module. Modules handle their state internally, while functionals need to have their state manually passed to them.

While most of the options passed to both versions are similar, there are still some differences. For example, the Linear module takes in the `in_channels` and `out_channels` as options, while the functional equivalent infers these from the state that is passed in. This prior work handled this by hardcoding which parameters are read from each layer as part of the parser code.

We can extract the options for a `call_function` node by looking at the entries in `node.`

`kwargs`. However, this is trickier in the case of modules. As we can see in Table 2.1, none of the nodes representing convolutional layers have any information on hyperparameters like stride or kernel size. This is because these options are constructor parameters to these `call_module` layers. We will discuss how to handle this in a more generalizable way in chapter 4.

Since the computation graphs of most non-recurrent models can be described as a directed acyclic graph (DAG), we simply perform a depth-first search (DFS) through the entire graph to get the order of computations. For each node, `node.args` is a tuple consisting of the nodes which provide the input to this node. We first iterate though all nodes in the graph and maintain a separate structure to map a node to a list of other nodes. Every time a node is encountered in `node.args`, the current node is added to that node entry's successors.

Chapter 3

Compiling StreamIt with LibTorch

While the prior work was sufficient to convert PyTorch graphs to StreamIt pipelines, it was still limited in 2 main ways: the PyTorch to StreamIt conversion relied on hardcoded hyperparameter extraction, and linking the generated code to the LibTorch library functions was not done. We will discuss how we generalized the former in chapter 4. The rest of this chapter talks about the changes we made to the first stage of the StreamIt compilation process.

3.1 Outline of Compilation Process

At a high level, a PyTorch model file will go through 3 main stages of compilation: conversion from a PyTorch model to a StreamIt pipeline, first-stage compilation to generated C++ code, and second-stage compilation to an optimized executable. This process is outlined in Figure 3.1.

In addition to the basic framework that already existed, we also generate the options and filter definitions at the first conversion stage. The filter definitions are used in the first stage of compilation to write the StreamIt pipeline layers. For example, a generated `conv2d` filter can be used to create a 2-dimensional convolutional layer in the StreamIt pipeline. The generated code on second-stage compilation will make use of the generated hyperparameter

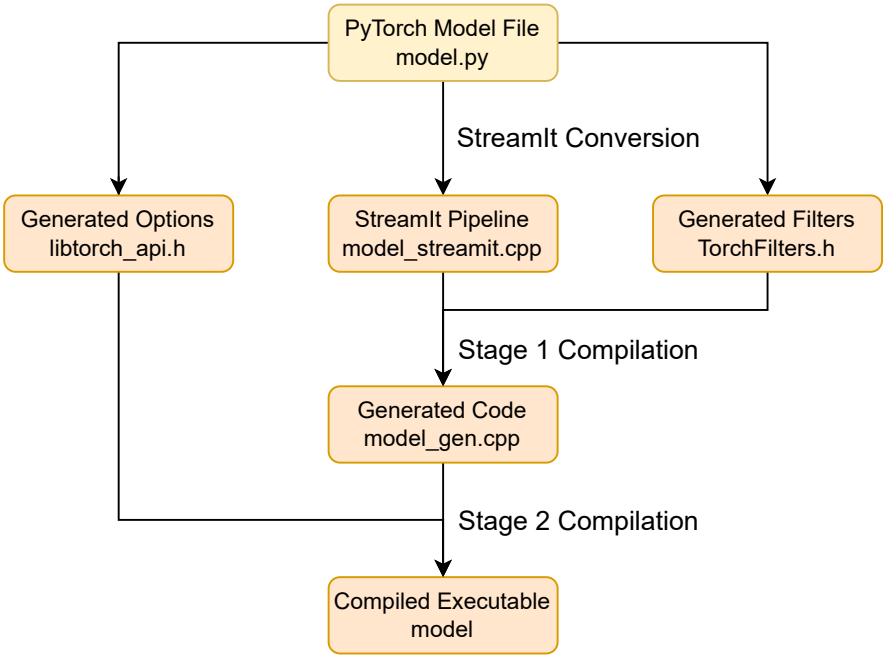

Figure 3.1: Compilation Process of a Model

options to generate an executable binary.

3.2 The StatefulFilter Construct

One main feature of StreamIt filters is that they don't have state and are hence called "pure" filters; they simply read inputs from the input stream and push their outputs to the output stream. In machine learning computation, most layers have some state associated with them like weights and biases. However, the existing StreamIt Filter definition is Stateless and doesn't allow an easy way to access these variables from the `work()` function. Hence, we create a new `StatefulFilter` construct that extends the Filter class and holds state.

Stateful filters extend regular StreamIt filters in 2 major ways: they store state, and they use this state in combination with the input channel to compute the output in the work function. From the implementation in Figure 3.2, we see that this class has two main methods: `init()` and `work()`. While all StreamIt constructs have a work function, Stateful

```
template <typename I, typename O, typename S>
class StatefulFilter : public Filter <I, O> {
  stateful_init_t<S> init_impl;
  stateful_work_t<I, O, S> work_impl;
  dyn_var<S> state;
  std::string state_name;
public:

  StatefulFilter(rate_t popRate, rate_t peekRate, rate_t pushRate,
   stateful_init_t<S> init_impl,
                 stateful_work_t<I, O, S> work_impl, std::string state) :
    init_impl(init_impl), work_impl(work_impl), state(builder::with_name(
   state)), state_name(state),
    Filter<I, O>(popRate, peekRate, pushRate) {}

  StatefulFilter(rate_t popRate, rate_t pushRate, stateful_init_t<S>
   init_impl, stateful_work_t<I, O, S> work_impl,
                 std::string state) :
    init_impl(init_impl), work_impl(work_impl), state(builder::with_name(
   state)), state_name(state),
    Filter<I, O>(popRate, pushRate) {}

  void init(dyn_var<HashMap<std::string, S>> states) {
    this->state = states.at(state_name);
  }

  void work() {
    work_impl(this->getInputBuffer(), this->getOutputBuffer(), this->state
   );
  }
};
```

Figure 3.2: The StatefulFilter class

filters uniquely have a user-customizable init function that is used to store state as a `builder::dyn_var` field.

The Stateful filter constructor takes in the name of the layer as exported in the pytorch weights file. This is used in combination with the BuildIt `with_name` construct to initialize the state field. On code generation, the states will be declared separately as global variables, and the `with_name` links these global variables as states for the layers.

In addition to the state field, the Stateful filter also accepts a `stateful_init_t` implementation, which is a function that copies its second argument into its first. This allows the program to customize the initialization of its state if needed. We currently do not make use

of this feature.

3.3 BuildIt Custom types

We also needed to add some BuildIt custom types to represent the types of hyperparameters and other inputs to each layer.

3.3.1 ExpandingArray

```
template<size_t D, typename T=int64_t>
struct ExpandingArray {
  std::vector<T> vals;
  ExpandingArray(): vals(std::vector<T>()) {}
  ExpandingArray(std::initializer_list<T> vals): vals(vals) {}
  ExpandingArray(T val): vals(std::vector<T>(val, D)) {}
  T operator[](size_t idx) const { return vals[idx]; }
};
```

Figure 3.3: The ExpandingArray type

LibTorch has its own types defined for convenience. The `ExpandingArray` type in particular is useful when passing options to layers. For example, a convolutional layer needs to be configured with the size of the kernel, padding, and stride. A 2-dimensional convolution needs 2 numbers for each of these, corresponding to the 2 axes, and it is often the case that both these numbers will be equal. The `ExpandingArray` type can hence be initialized with either a single number that is replicated across all its dimensions or a vector of numbers.

The BuildIt custom type definition reflects this by also providing both types of constructors.

We also need a structure in the second stage of BuildIt compilation to describe an ExpandingArray. However, at this stage, we already know all the elements of the `ExpandingArray`, so a simple container that behaves like a C++ `std::vector` is sufficient. The LibTorch `ExpandingArray` type is constructible from an `std::vector`.

We still need to write BuildIt code to copy over each element individually from the `ExpandingArray` on the StreamIt side to this new `std::vector` container. We accomplish this by writing a BuildIt static loop in the `copy_expanding_array` function that copies over each element of the array.

```
template<typename T>
struct vector: builder::custom_type<T> {
  static constexpr const char* type_name = "std::vector";
  builder::dyn_var<void(T)> push_back = builder::as_member("push_back");
};

template <size_t D, typename T=int64_t>
void copy_expanding_array(builder::dyn_var<vector<T>>& x, ExpandingArray<D
    , T> const& w) {
  for (builder::static_var<T> i = 0; i < D; i++) {
    x.push_back(w[i]);
  }
}
```

Figure 3.4: Copying an ExpandingArray into a vector

In Figure 3.4, the `copy_expanding_array` function can be used to copy elements from the `ExpandingArray` w to the `vector` x. The generated code for such a call is given in Figure 3.5. In this example, the `ExpandingArray` w was initialized with the values {1, 1}. As is evident, the generated code only has the `std::vector` type, and all values in the array were expanded out at compile-time.

```
// written code
builder::dyn_var<vector<int64_t>> x;
ExpandingArray<2> w(1);
copy_expanding_array(x, w);

// generated code
std::vector<long int> var12;
var12.push_back(1ll);
var12.push_back(1ll);
```

Figure 3.5: Generated code for copying an ExpandingArray into a vector

3.3.2 Tensor

```
class Tensor: public builder::custom_type<> {
public:
  static constexpr const char *type_name = "torch::Tensor";
  dyn_var<bool()> contiguous = builder::as_member("contiguous");
  dyn_var<Tensor(int)> split = builder::as_member("split");
  dyn_var<Tensor()> transpose = builder::as_member("transpose");
  dyn_var<Tensor()> view = builder::as_member("view");
  // ...
};
```

Figure 3.6: The BuildIt Tensor type

LibTorch uses the ATen library's Tensor class to represent the weights and activations. This class also has member functions like `transpose()` and `contiguous()` that perform certain operations on the Tensor.

The Tensor custom type is implemented to have all the fields and methods a typical LibTorch Tensor has, but in the form of BuildIt specifications. The fields in the custom Tensor type can be used to write the filter `work()` function code. After all stages of compilation, these calls will be converted to the LibTorch Tensor type's method calls.

3.3.3 LayerState

```
struct LayerState : builder::custom_type<> {
  static constexpr const char* type_name = "LayerState";
  builder::dyn_var<libtorch::Tensor> weight = builder::as_member("weight")
    ;
  builder::dyn_var<libtorch::Tensor> bias = builder::as_member("bias");

  // additional state for batch norm
  builder::dyn_var<libtorch::Tensor> mean = builder::as_member("mean");
  builder::dyn_var<libtorch::Tensor> variance = builder::as_member("
    variance");
};
```

Figure 3.7: LayerState struct definition

This custom type is used to store the weights and biases of LibTorch model layers. The state of a layer in a StreamIt ML pipeline is represented by weight and bias tensors. However, there are certain layers, like the batch normalization layer, which keep track of additional

state. The batch normalization module in PyTorch keeps a running mean and variance of all batches that it encountered during training. At the time of inference, these values are then used to normalize the inputs. Since these are also part of the state of certain layers, they are included in the LayerState struct.

Each layer has its own set of hyperparameter options, as discussed in the ExpandingArray section. These options are initialized once but used many times, so they should be stored in the LayerState type too. Each layer in LibTorch that takes options has a separate struct defined for that particular layer. For example, the options for the 2-dimensional convolutional layer is given by the `torch::nn::functional::Conv2dFuncOptions` struct, while those for the dropout layer is given by the `torch::nn::functional::DropoutFuncOptions`. Since these options are for different layers, they don't have a common base class. This would require us to make the options field into a template type parameter in the LayerState struct and instantiate the LayerState with the options type corresponding to that layer type. We do not tackle this problem in this thesis work. Instead, we simply create a new options struct just before a layer is called during the forward pass and immediately pass it into the call.

3.4 Generated code

The generated code for all models compiled via StreamIt are implementations of two common header file functions: `init()` and `forward()`. This way, the interfaces for all models are common, and we can swap out the C++ implementations if the model needs to be changed.

```
1 void init (std::unordered_map<std::string, LayerState> state_dict);
2
3 void forward(streamit::deque<torch::Tensor>* &arg1, streamit::deque<torch
    ::Tensor>* &arg2);
```

Figure 3.8: init() and forward() declarations

3.4.1 init()

We need a way to initialize the weights and biases of each stateful Filter in the StreamIt machine learning code from an external source like a saved model file. This was implemented through an `init()` function. The init function takes in a transformed state-dictionary of the model's weights and biases and initializes each layer with their corresponding parameters. This dictionary maps the fully-qualified identifier of each layer to a LayerState struct available at runtime with the weight and bias as fields.

```
for(static_var<int> k = 0; k < repeats; k++) {
  for(static_var<int> i = 0; i < scheduleSteady.size(); i++) {
    auto* filter = dynamic_cast<StatefulFilter<I, O, S>*>(scheduleSteady[i
   ]);
    if (filter != nullptr) {
      filter->init(state_dict);
    }
  }
}
```

Figure 3.9: BuildIt code for init()

BuildIt expands the static for loop into a sequence of calls to `StatefulFilter init()` functions, each of which initializes its state by reading from the state-dictionary.

3.4.2 forward()

The forward function performs the actual forward-pass inference computation of the model. This function contains a while-loop that continuously reads from the input channel and writes the computed result to the output channel after doing one inference pass.

Similar to the `init()` function, BuildIt expands the static for loop in the case of the `forward ()` function. However, it retains a while loop that is not expanded at compile-time, since there can be more than one inference request, and this number is not known at compile-time.

```
while (1) {
  for(static_var<int> k = 0; k < repeats; k++) {
    for(static_var<int> i = 0; i < scheduleSteady.size(); i++) {
      scheduleSteady[i]->work();
    }
  }
}
```

Figure 3.10: BuildIt code for forward()

3.5 LibTorch API

PyTorch layers in Python are constructed in the constructor method of a module, with hyperparameters such as kernel size and stride passed in directly as constructor parameters. Since these constructor parameters are generally passed in as keyword arguments, they can be ordered in any way, or some can even be omitted in favor of default values. The `forward()` method of the module can then use these constructed submodules in its inference pass.

Since the LibTorch API is written in C++, parameters cannot be reordered in the constructor. Because of this, the hyperparameters of each layer are passed in as a C++ structure constructed using the builder pattern, which we will call "options" in this thesis.

```
auto options = F::Conv2dFuncOptions()
      .stride(stride)
      .padding(padding)
      .dilation(dilation)
      .groups(groups)
      .bias(bias);
```

Figure 3.11: Creating an options instance for a Conv2d layer

3.5.1 Modules vs Functionals

In addition to the difference above, we also need to convert all modules to function calls. Almost all built-in PyTorch modules have functional equivalents in `torch.nn.functional`.

PyTorch modules handle their state internally. This includes the layers' weights and biases. While this is useful from an abstraction point-of-view, hiding away state robs us of

potential optimization opportunities. Modules also compute the backward gradients unless explicitly turned off, which is an additional unnecessary overhead for inference-only models.

The functional equivalents of modules need to have their state passed to them on every function call. For example, a 2-dimensional convolutional functional in Figure 3.12 needs a `weight` instance passed to it.

```
torch::Tensor res = F::conv2d(input, weight, options);
```

Figure 3.12: Computing a Conv2d result

These functionals are stateless themselves, and a given set of inputs, weights, and biases will always produce the same outputs. Thus, converting module calls to functionals allows us more avenues of optimization in the future by retaining more control over the computation and state.

3.6 MNIST Model Compilation process

We will use the example of an MNIST model to illustrate the build process starting from a StreamIt pipeline definition to the generated `init()` and `forward()` code. The StreamIt pipeline definition for the model is given in Figure 3.13.

```
auto p = pipeline<libtorch::Tensor, libtorch::Tensor>([&](auto pipe) {
  pipe->add(libtorch::conv2d("conv_layers_0", {1, 1}, {0, 0}, {1, 1}, 1, "
    zeros", {0, 0}, {3, 4}));
  pipe->add(libtorch::relu("conv_layers_1", false));
  pipe->add(libtorch::conv2d("conv_layers_2", {1, 1}, {0, 0}, {1, 1}, 1, "
    zeros", {0, 0}, {3, 3}));
  pipe->add(libtorch::relu("conv_layers_3", false));
  pipe->add(libtorch::conv2d("conv_layers_4", {1, 1}, {0, 0}, {1, 1}, 1, "
    zeros", {0, 0}, {3, 3}));
  pipe->add(libtorch::relu("conv_layers_5", false));
  pipe->add(libtorch::flatten("fc_layers_0", 1));
  pipe->add(libtorch::linear("fc_layers_1"));
});
```

Figure 3.13: StreamIt pipeline code for an MNIST model

Here, each `pipe->add()` call takes as argument a filter that has the corresponding layer operation as its `work()` function. This way, an input flowing through a pipeline gets transformed into the inference result. The filter definition for the Linear layer is given in Figure 3.14.

```
builder::dyn_var<libtorch::Tensor(libtorch::Tensor, libtorch::Tensor,
    libtorch::Tensor)> _linear = builder::as_global("F::linear");

std::shared_ptr<Stream<libtorch::Tensor, libtorch::Tensor>>
linear(std::string node_name) {
  auto filter = makeStatefulFilter<libtorch::Tensor,libtorch::Tensor,
    LayerState>(1, 1,
     [](auto state, auto init_state) {
       state = init_state;
     },
     [=](auto in, auto out, auto state) {
       out->push(_linear(in->pop(), state.weight, state.bias));
     },
     node_name);
  filter->setName(node_name);
  return filter;
}
```

Figure 3.14: StreamIt code for a Linear layer

Note that we create a `StatefulFilter` instance for the `Linear` layer. This is because the layer has an associated weight and bias. The work function calls `_linear()`, which is defined as a BuildIt `as_global` construct, by passing in `state.weight` and `state.bias` as the appropriate arguments. This signifies that the definition for this type will be provided during the second stage of the multi-stage compilation process as the `F::linear()` function.

After the second stage of the compilation process, the linear layer of the MNIST model generates the code in Figure 3.15. Here, `var8[0]` contains the tensor output from the layer before the linear layer, and the output of the model's forward pass is pushed onto an output deque `outputWorker0`.

```
torch::Tensor var64 = var8[0];
torch::Tensor var65 = F::linear(var64, fc_layers_1.weight, fc_layers_1.
    bias);
var9[0] = var65;
torch::Tensor var69 = var9[0];
outputWorker0.push(var69);
```

Figure 3.15: Generated MNIST Linear layer forward() code

Chapter 4

PyTorch Graphs to StreamIt

We now have a way to write machine learning models as StreamIt pipelines which can generate working code, and we talked about how PyTorch models can be converted to these StreamIt pipelines in chapter 2. We extend this framework to make adding new layer types to the PyTorch converter easier.

4.1 Extracting options from `torch.fx` nodes

As we discussed earlier, the constructor parameters are not readily available for PyTorch modules. We retrieve these values by looking for the option names in `module.__constants__` for that particular module. It is possible that certain options might be extraneous, so we deal with these cases by allowing a manual configuration override that we discuss later in this chapter.

4.2 Autogenerating necessary types and functions

In the previous chapter, we discussed that the prior work used manually written filter-making functions for each type of layer. While these functions can be written by hand, it is better to automate their generation to the maximum extent possible, since their basic structures

are similar to each other. To aid this, we parse each node in the PyTorch IR graph to a `StreamitFilter` type, which can be an instance of either a `StreamitLayer` for "`call_function`", and "`call_module`" node types, or a `StreamitGetAttr` for "`get_attr`" or "`call_module`" node types.

4.2.1 StreamitFilter

The `StreamitFilter` type has three main methods that each of its subtypes implement: `generate_buildit_decl`, `generate_options_builder`, and `generate_streamit_filter`. We will illustrate these using a Dropout layer as an example.

generate_buildit_decl()

This method generates the BuildIt type information required for the first stage compilation. This includes `builder::dyn_var` declarations for the LibTorch layer functional and the LibTorch options struct.

```
constexpr char dropout_options_name[] = "F::DropoutFuncOptions";

using dropout_options_t = typename builder::name<dropout_options_name>;

builder::dyn_var<dropout_options_t(float,bool,bool)> make_dropout_options
    = builder::as_global("make_dropout_options");

builder::dyn_var<libtorch::Tensor(libtorch::Tensor, libtorch::Tensor,
    libtorch::Tensor, dropout_options_t)> _dropout = builder::as_global("F
    ::dropout");
```

Figure 4.1: Sample BuildIt type declaration

The generated code snippet in Figure 4.1 creates two main BuildIt dynamic variables: `make_dropout_options` and `_dropout`. `_dropout` corresponds to the actual filter implementation provided by the LibTorch API, `torch::nn::functional::dropout`. We abbreviate `torch::nn::functional::` to simply `F::` in these examples.

`make_dropout_options` creates the options struct populated with the hyperparameters of the layer. We will discuss the actual implementation of this function in the next part.

generate_options_builder()

This method generates a function that constructs the options struct for a given layer that takes configurable hyperparameters.

```
1
2 inline F::DropoutFuncOptions
3 make_dropout_options(float p, bool training, bool inplace) {
4     return F::DropoutFuncOptions()
5         .p(p)
6         .training(training)
7         .inplace(inplace);
8 }
```

Figure 4.2: Sample BuildIt type declaration

The generated code snippet in Figure 4.2 constructs the LibTorch options struct, `F::DropoutFuncOptions`, corresponding to the dropout layer. This header file will be linked during the second-stage compilation process of the StreamIt pipeline.

generate_streamit_filter()

This method generates the filter-making function that returns either a `Filter` or a `StatelessFilter` with the `work()` function as the layer computation.

The generated function in Figure 4.3 returns a Streamit StatefulFilter instance with the work function set to the dropout layer's operation. This header file is linked during the first-stage first stage compilation of the StreamIt pipeline.

4.2.2 The JSON Configuration File

We discussed previously that there are certain differences in the API of modules and functionals. There are also small differences between the PyTorch and LibTorch versions of certain layer types. In order to provide an easy way to tweak the generated filters and options manually if needed, we implement a configuration file and update it accordingly.

```
std::shared_ptr<Stream<libtorch::Tensor, libtorch::Tensor>>
dropout(std::string node_name, float p, bool training, bool inplace) {
  auto filter = makeStatefulFilter<libtorch::Tensor,libtorch::Tensor,
   LayerState>(1, 1,
      [](auto state, auto init_state) {
        state = init_state;
      },
      [=](auto in, auto out, auto state) {
        auto options = make_dropout_options(p, training, inplace);
        out->push(_dropout(in->pop(), state.weight, state.bias, options));
      },
      node_name);
  filter->setName(node_name);
  return filter;
}
```

Figure 4.3: Sample BuildIt type declaration

We use a JSON file to store information about the filters and options from other models. This file contains an entry for each type of filter encountered during the PyTorch to StreamIt conversion process. The entries are auto-generated if they are not already present in the file. We also use this file to override the options parsed from the PyTorch model in the case of differences between the PyTorch and LibTorch versions of the same layer. The format of an entry is shown in Figure 4.4.

```
{
"dropout": {
  "body_overrides": {},
  "dests": 1,
  "filter_name": "dropout",
  "libtorch_name": "F::dropout",
  "options": [
    "p float",
    "training bool",
    "inplace bool"
  ],
  "options_name": "F::DropoutFuncOptions",
  "srcs": 1,
  "state": 2,
  "type": "call_module"
},
}
```

Figure 4.4: A sample Dropout layer JSON entry

- `srcs` and `dests`: The number of layers pushing outputs into this layer and the number of layers reading from the output of this layer respectively.
- `filter_name`: The name of the generated StreamIt filter-making function.
- `libtorch_name`: The name of the LibTorch API layer.
- `options`: A list of strings, each corresponding to a hyperparameter. Each string contains two parts separated by a space. The first part is the name of the parameter, while the second part is the C++ datatype. Since identifiers cannot contain spaces, the type information can contain as many spaces as needed.
- `options_name`: The name of the LibTorch API options struct corresponding to this layer. If the options field is present but this field is absent, the options are directly passed to the layer call.
- `state`: The number of distinct tensors used to represent the state associated with this layer.
- `type`: The opcode of the layer.
- `body_overrides`: This field is used if the actual LibTorch implementation of a layer is very different from the PyTorch version. If this object contains the entry `"work": true`, then the StreamIt filter-making function and its related BuildIt types are not generated. Hence, we can write our own implementation of the filter-maker in a different header file that isn't auto-generated. This field can also contain the entry `"options": true`, in which case the options builder is not generated.

4.3 ResNet Model Conversion

To illustrate the conversion from a PyTorch model to a StreamIt pipeline definition, we'll look at a section of a more complicated ResNet18 model from the torchvision Python module.

Since the code is not readily available, Figure 4.5 contains the traced code outputted by `torch.fx`. After performing a DFS of the PyTorch graph, we generate StreamIt code along with a DOT graph. A small snippet of both are shown in Figure 4.6 and Figure 4.7.

```
# ...
layer1_0_conv1 = getattr(self.layer1, "0").conv1(maxpool)
layer1_0_bn1 = getattr(self.layer1, "0").bn1(layer1_0_conv1);
    layer1_0_conv1 = None
layer1_0_relu = getattr(self.layer1, "0").relu(layer1_0_bn1);
    layer1_0_bn1 = None
layer1_0_conv2 = getattr(self.layer1, "0").conv2(layer1_0_relu);
    layer1_0_relu = None
layer1_0_bn2 = getattr(self.layer1, "0").bn2(layer1_0_conv2);
    layer1_0_conv2 = None
add = layer1_0_bn2 + maxpool;  layer1_0_bn2 = maxpool = None
layer1_0_relu_1 = getattr(self.layer1, "0").relu(add);  add = None
#...
```

Figure 4.5: PyTorch ResNet18 torch.fx traced code snippet

ResNet models work on the basis of residual activations [4]. This means that, instead of learning a function $f(x)$, a block learns the difference $f(x) - x$. This solves the problem of exploding and vanishing gradients, a phenomenon where the weights of a neural network either vanish to 0 or explode to a large value during training. This branching structure of a ResNet model thus necessitates the use of a StreamIt SplitJoin construct, as we can see from both figures. A Duplicate SplitJoin is generated here because we need to make two copies of the tensor x.

```
//...
pipe->add(
  duplicateSplitJoin<libtorch::Tensor, libtorch::Tensor>([&](auto sj) {
    sj->add(1,
      pipeline<libtorch::Tensor, libtorch::Tensor>([&](auto pipe) {
        pipe->add(libtorch::conv2d("layer1_0_conv1", {1, 1}, {1, 1}, {1,
   1}, 1, "zeros", {0, 0}, {3, 3}));
        pipe->add(libtorch::batchnorm2d("layer1_0_bn1", true, 0.1, 1e-05))
   ;
        pipe->add(libtorch::relu("layer1_0_relu", true));
        pipe->add(libtorch::conv2d("layer1_0_conv2", {1, 1}, {1, 1}, {1,
   1}, 1, "zeros", {0, 0}, {3, 3}));
        pipe->add(libtorch::batchnorm2d("layer1_0_bn2", true, 0.1, 1e-05))
   ;
      })
    );
    sj->add(1, libtorch::identity<libtorch::Tensor>());
  })
);
pipe->add(libtorch::add("add"));
pipe->add(libtorch::relu("layer1_0_relu_1", true));
//...
```

Figure 4.6: PyTorch ResNet18 generated StreamIt pipeline code snippet

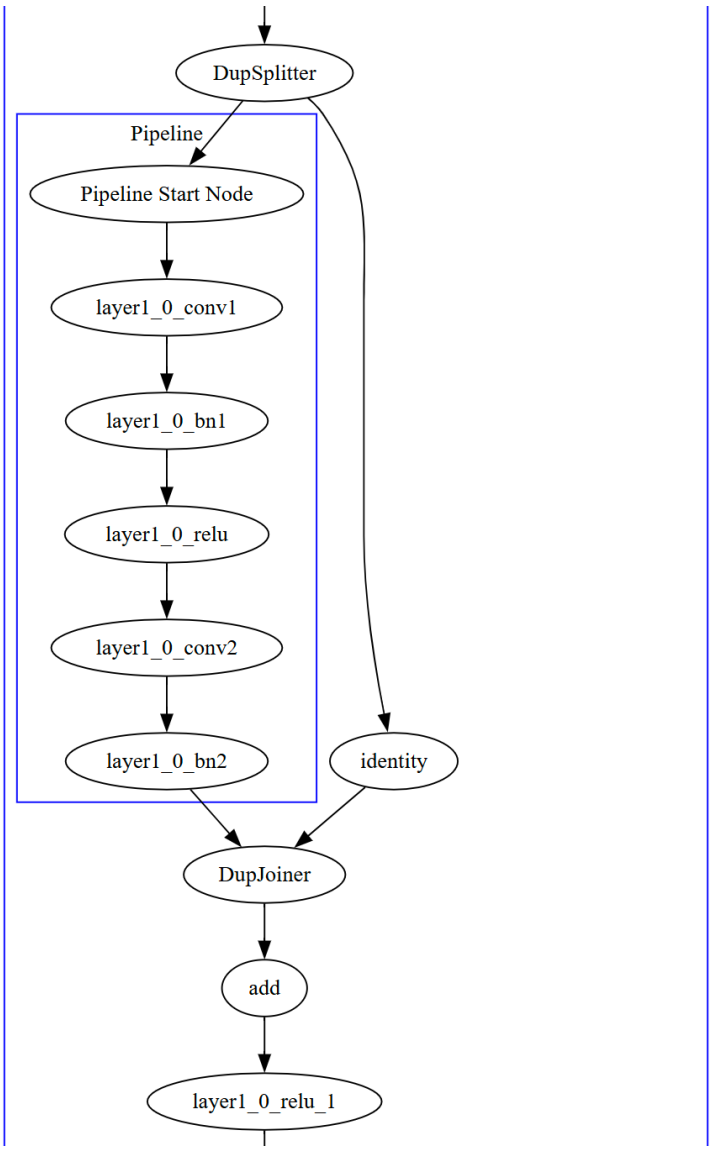

Figure 4.7: A section of the DOT graph of ResNet18 model in StreamIt

Chapter 5

Experiments and Future Work

The performed experiments measure the difference in runtimes of a PyTorch model, an inference-optimized version of the same model in LibTorch with a call to `torch::jit::optimize_for_inference`, and the StreamIt-compiled version of the model generated by the methods put forth in this thesis.

We use the version of ResNet with the ImageNet1k v1 weights provided by the torchvision package to run the experiments. All the experiments are performed on Lanka v3 nodes. Lanka is a 24-node Intel Xeon E5-2695 v2 @ 2.40GHz Infiniband cluster with two sockets per node, each with 12 cores, for a total of 576 cores and a theoretical peak computational rate of 11059 GFlop/s. Each node has 128GB of memory, of which 120GB is safely usable.

Each experiment involves passing 100 randomly generated tensors of shape 1 x 3 x 224 x 224 to each model and measuring the overall time taken for these 100 inference queries. These images are of dimensions 224 x 224, with 3 channels for color. The images are input in batches of 1.

We vary the number of layers of the ResNet model among 18, 34, 50, 101, and 152, which are the sizes used by the authors of the ResNet paper [4]. Each block in a ResNet18 and ResNet334 model has 2 3x3 convolutional layers. ResNet18 has 8 blocks, while ResNet34 has 16 blocks. The 3x3 here refers to the kernel size. In ResNets 50 onwards, there are 3

convolutional layers per block of sizes 1x1, 3x3, and 1x1. ResNet 50 has 16 blocks, while the bigger models have more blocks. The descriptions of blocks in the ResNet models are summarized in Figure 5.1.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 5.1: ResNet Model Architectures[1]

Version / Model Size	18	34	50	101	152
StreamIt runtime	7.78943	14.0211	18.1437	31.3938	45.0847
PyTorch runtime	6.8626	12.7964	14.1468	25.4368	36.6504
LibTorch runtime	6.86867	12.3459	14.0023	25.0262	36.03

Table 5.1: Results of runtime experiments (runtime in seconds)

5.1 Discussion of Results

The results in Figure 5.2 and Table 5.1 show that both the PyTorch and LibTorch versions of the model outperform the StreamIt model for all model sizes, while optimizing for inference in the LibTorch version did not change the latencies by much. Since the base PyTorch model with gradient computations turned off and in `eval()` mode outperforms the StreamIt generated code, we can conclude that this difference is due to the overhead of copying tensors.

[1]Credit: The ResNet paper [4]

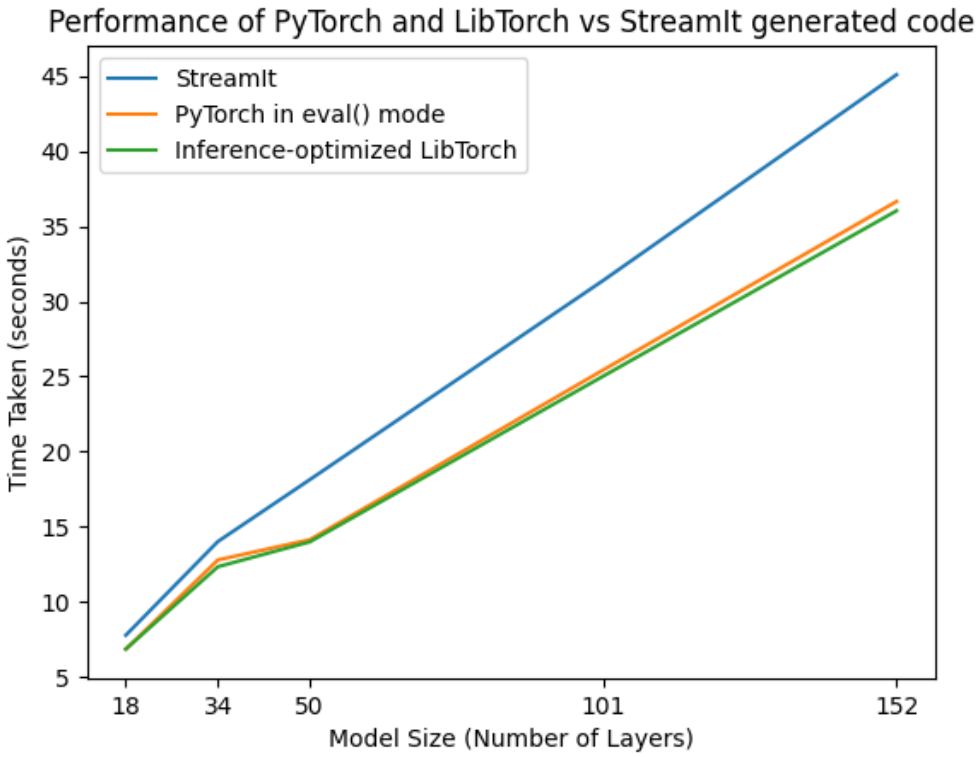

Figure 5.2: Performances of the PyTorch and StreamIt ResNet models

It is also clear that all 3 versions of the model show a higher latency for larger models. This is because increasing the number of layers linearly increases the amount of computation done for a single inference pass.

It is also interesting to note that the difference in latencies between the StreamIt and PyTorch versions becomes more prominent when the layer size is 50 or up. The StreamIt version of the model linearly increases with layer size, while the PyTorch and LibTorch versions seem to stabilize in latency between the 34-50 region. Since the number of layers per block changes from 2 to 3 in this region, this lack of meaningful difference in latencies can be attributed to a change in model architecture. Adding more convolutional layers makes the latencies scale differently than adding more blocks which adds other layers like BatchNorm2d and ReLU. This means that the latency of convolutional layers has more of an impact in the StreamIt pipeline.

This thesis work puts forth a way to convert PyTorch models to static StreamIt pipelines.

As future work, several optimization passes can be added to the generated static StreamIt pipeline to make it faster than the PyTorch and LibTorch models. The generated code could also be made more optimized with lesser instances of copying the elements in the stream, possibly by using references to the tensors instead.

5.2 Future Work

The StatefulFilter class can be used to store arbitrary state information associated with the filter. Since we only need to initialize the layer's options once during initialization of the LibTorch layer, storing the options as part of the state should lead to performance improvements in the generated code. However, this is not handled as past of this thesis work.

The StreamIt compiler can be extended to larger models like GPT2. However, there are still some issues with the implementation of certain layers and functionality that need to be addressed. The current implementation of the PyTorch to StreamIt conversion does not handle tensor constants. These are predefined tensors stored as attributes of the module. These would ideally need to be constants in the StreamIt static graph, but there is no easy way to determine whether an attribute is a tensor constant or simply a property of a tensor.

Another issue is that the implementation currently only allows tensor instances to flow through the StreamIt graph. However, some intermediate data can be of other datatypes too which we don't handle as of now. A solution to this is by creating a superclass for all objects that could flow through the graph, but this will eliminate the possibility of certain optimizations that rely on the type of the object.

A major benefit of a StreamIt computation graph is the possibility of scheduling work on machines with multiple computation units in a distributed pipeline-parallel fashion. While we do not explore this in this work, this is the potential next step in the work presented in this thesis.

5.3 Conclusion

In this work, we have implemented a PyTorch to StreamIt compiler that can generate code through an optimizable static graph. According to the results, the generated StreamIt pipeline has a higher latency than the PyTorch version of that model without any optimizations. However, there is still room for more optimizations to be added to the StreamIt pipeline.

References

[1] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024. doi:10.1145/3620665.3640366. URL https://pytorch.org/assets/pytorch2-2.pdf.

[2] Ajay Brahmakshatriya and Saman Amarasinghe. Buildit: A type-based multi-stage programming framework for code generation in c++. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 39–51. IEEE, 2021.

[3] Kaustubh Dighe. Fast multistage compilation of machine learning computation graphs. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2024. URL https://commit.csail.mit.edu/papers/2024/Dighe_MEng_Thesis.pdf.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. URL https://arxiv.org/abs/1512.03385.

[5] Ilya Lavrenov, Ilya Churaev, Roman Kazantsev, Vladimir Paramuzov, Maxim Vafin, Irina Efode, Sebastian Golebiewski, Pawel Raasz, Karol Blaszczak, Mateusz Tabaka, Vitaliy Urusovskij, Vladislav Golubev, Anastasia Kuporosova, Katarzyna Mitrus, Sergey Shlyapnikov, Ivan Tikhonov, Alexandra Sidorova, Maciej Smyk, Anastasiia Pnevskaia, Taylor Yeonbok Lee, Mikhail Ryzhov, Oleg Pipikin, Edward Shogulin, Michael Nosov, Egor Duplenskii, Roman Lyamin, Tomasz Jankowski, Alina Kladieva, and Andrei Kashchikhin. openvinotoolkit/openvino. URL https://github.com/openvinotoolkit/openvino.

[6] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction: 11th International Conference, CC 2002 Held as Part of the Joint European Conferences on Theory and Practice of*

Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings 11, pages 179–196. Springer, 2002.