MASTER'S THESIS 2024

Pipelined Context Switching for Deep Learning Models

Fredrik Horn Dannert

ISSN 1650-2884
LU-CS-EX: 2024-53

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-53

Pipelined Context Switching for Deep Learning Models

Pipelined Context Switching för Djupa Inlärningsmodeller

Fredrik Horn Dannert

Pipelined Context Switching for Deep Learning Models

Fredrik Horn Dannert
fredrik.dannert@gmail.com

August 23, 2024

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Jonas Skeppstedt, jonas.skeppsted@cs.lth.se

Examiner: Michael Doggett, michael.doggett@cs.lth.se

Abstract

This thesis investigates the efficiency of pipelined context switching for deep learning models, specifically within single GPU environments. The motivation stems from the increasing computational demands of training and deploying deep learning models, necessitating optimized resource utilization to reduce costs and improve performance. The thesis evaluates an existing pipelined context switching system, originally implemented in Pytorch-1.3.0, and ports it to Pytorch-2.1.0. An evaluation is conducted on three different Nvidia GPUs: T4, A30, and A40, using models such as Bert_Base, Inception_V3, and Resnet152.

The work presented highlights the benefits of pipelined context switching in meeting service level objectives (SLOs) for models sharing limited single GPU resources, and pipelined context switching can reduce the incurred overhead by up to 10x from transmitting model parameters over PCIe to the device, showing promise for future work.

Keywords: Deep Learning, Pipelined Execution, DNN Context Switching

Acknowledgements

I want to thank my supervisor Jonas Skeppstedt for his feedback and help on my thesis. I would also like to thank my colleagues at Huawei, Gingfung Matthews for tremendous support in problem solving and bouncing ideas. Amardeep Mehta for helping me with test cases, feedback and supervising my time at Huawei. Adam Barker for giving me the opportunity to write my thesis with the Systems team.

Contents

1 Introduction **7**
1.1 Problem Statement 8
1.2 Research Questions 8
1.3 Contribution 8
1.4 Background 8
1.4.1 Deep Learning Models 8
1.4.2 Execution of Deep Neural Network Models 11
1.4.3 NVIDIA GPUs 14
1.4.4 Pytorch 15
1.4.5 Inter Process Communication 16
1.5 Related Work 17
1.5.1 Layer Aware Pipelining 17
1.5.2 Batch Parallelism and Pipelining 17
1.5.3 Microsecond Inference Preemption 18
1.5.4 Resource Estimation and Scheduling, Grouping Algorithm 18

2 Approach **19**
2.1 Method 19
2.1.1 Architecture Overview 20
2.1.2 Replication of Experiments 21
2.2 Implementation 21
2.2.1 Port to Pytorch 2.1 21
2.2.2 CUDACachingAllocator in Pytorch 1.3 and 2.1 21
2.2.3 Binding to Python 27
2.2.4 Workers and Scheduler Memory Sharing 27
2.2.5 Pipelined Execution 28

3 Evaluation **31**
3.1 Experimental Setup 31
3.2 T4 Comparison Results 33

3.2.1 1.3.0 and 2.1.0 Comparisons 33
3.3 T4, A30 and A40 Results for Pytorch 2.1.0 37

4 Discussion and Conclusion **41**
4.1 Evaluation of Results 41
4.1.1 Pytorch 1.3.0 and 2.1.0 implementations 41
4.1.2 Pipelined Context Switching and Linear Transmission 42
4.1.3 Pipelined Execution to Meet SLO Requirements 42
4.2 Problem Summary 42
4.2.1 Research Question One 43
4.2.2 Research Question Two 43
4.3 Contribution 43
4.4 Future Work 44
4.4.1 Preempting Training Jobs 44
4.4.2 Framework Agnostic Implementation 44

Bibliography **45**

Chapter 1
Introduction

The adoption of Deep Learning (DL) models has been steadily increasing, fueled by the ever-growing capabilities of artificial intelligence (AI) and machine learning technologies. These models are integral to a wide array of applications ranging from natural language processing and computer vision to autonomous systems and predictive analytics. Companies across various industries harness the power of deep learning to extract meaningful insights from their vast datasets, enabling the creation of innovative products and services that were previously unattainable. This explosive growth in AI is exemplified by leading organizations such as OpenAI with ChatGPT, Anthropic with Claude, and Google DeepMind, among others. These companies have revolutionized the accessibility of advanced AI models, making sophisticated Generative Pre-trained Transformer (GPT) models that can understand and communicate text in a way that closely resembles human interaction[17]. These models are also provided for free or on a subscription basis.

OpenAI's ChatGPT, for instance, has become a cornerstone in the AI landscape, achieving over 200 million active monthly users as of June 2023[16]. This widespread adoption highlights the significant demand for AI-driven services, which are expected to deliver seamless and efficient user experiences. The challenge for these companies lies not only in meeting the high user expectations but also in managing the substantial computational resources required to train and deploy these models. Training state-of-the-art deep learning models involves extensive use of GPU clusters, which are both expensive and resource-intensive. Consequently, optimizing GPU resource utilization becomes paramount to reducing operational costs and improving efficiency. This optimization ensures that companies can continue to innovate and provide high-quality AI services without incurring prohibitive costs.

In this context, our research seeks to evaluate an existing system for pipelined context switching aimed at enhancing the efficiency of single GPU utilization in deep learning frameworks. The system should seek to enable the simultaneous sharing of GPU resources among multiple models that may not fit in memory concurrently while minimizing the overhead associated with model switching to meet client *service level objectives* (SLO). Service level objectives refer to the maximum delay an inference request is allowed to have, and staying within

those bounds is paramount to certain services that are stronly affected by delay in response time.

1.1 Problem Statement

We seek to find a system that performs a layer-aware execution strategy for a single GPU. This system has to be able to integrate with modern Deep Learning System frameworks like Pytorch or Tensorflow. The system at hand needs to be able to share single GPU resources between multiple models that might not fit in memory simultaneously. The caused overhead from switching models in and out of memory needs to be minimized as we aim to meet all *service level objectives* of our inference requests.

1.2 Research Questions

With a look into an existing solution of pipelined context switching, we want to explore the following:

1. Can pipelined execution of *Deep Neural Networks* still be viable for more recent (Ampere) Nvidia GPUs with regards to existing implementations? Moreover, do we see any improvements in GPU utilization for these devices?

2. Is pipelined context switching an execution strategy that can help minimize transmission overheads of model parameters to the GPU while still meeting *service level objectives*.

We will evaluate multiple GPU accelerators using our own implementation and see how they fare comparatively to the older implementation.

1.3 Contribution

The contribution will be a better understanding of how a system, that utilizes pipelined context switching to swap models in and out of memory, performs on newer generations of Nvidia GPUs and how it can be implemented. After this project, a reader should have a better understanding of how pipelined execution and transmission can be used to reduce swapping overhead while simultaneously addressing SLO requirements.

1.4 Background

1.4.1 Deep Learning Models

Deep learning, a subset of machine learning, is distinguished by its use of artificial neural networks (ANNs) with multiple layers, or "deep" architectures, to model complex patterns in data. This section outlines the fundamental components and processes that constitute deep

learning, offering insights into capabilities for inference and learning from vast amounts of data.

A ***Deep Neural Network*** (DNN) is a computational model with multiple layers, with capabilities for pattern recognition in data with many layers of abstraction[11]. Generally, a deep neural network comprises 3 types of layers; the *input* layer, the number of *hidden* layers, and the *output* layer. Figure 1.1

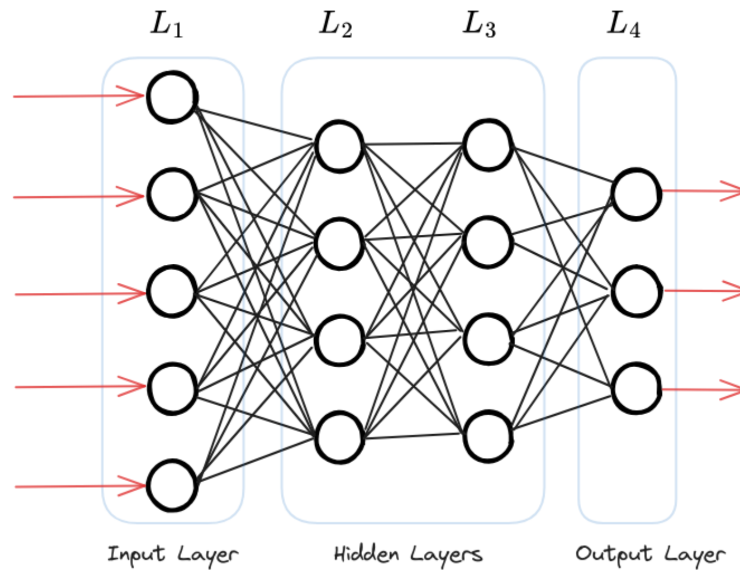

Figure 1.1: DNN with 4 layers. 1 input layer, 2 hidden layers, and 1 output layer.

Forward Propagation or forward pass, is the process in which a neural network makes a prediction given input data. In the forward pass, the data flow passes from the input layer, L_1 to the output layer L_N, where for each layer a prediction on the data is performed by using current weights and biases.

The ***Loss Function*** measures the difference between the network's prediction and the actual target values.

Backward Propagation allows the neural network to correct for its misprediction and is the core of where the learning is happening. The gradient of the loss function is calculated with respect to each weight in the network. The gradient is calculated starting from layer L_N and propagating to layer L_1. Lastly, the weights are updated and a step is taken in the opposite direction of the gradient. Put together, forward propagation, calculating the loss function, backward propagation, and stepping the optimizer is what is called an *epoch* or *iteration* of the DNN.

DNN Architecture

Definition 1 (Topological Ordering)**.** A topological ordering of a directed graph $G(V, E)$ is a linear ordering of its vertices V such that for every directed edge $(u, v) \in E$ from u to v, u will come before v in the ordering.

DNN architectures have a property of its layers being *topologically ordered*. This implies the layers $L_1, L_2 \cdots L_i, \cdots L_{N-1}, L_N$ all have an ordering of computation and subsequently

a dependency relation to respect. Topological ordering in DNNs states that a layer L_i will perform computation before subsequent layers $L_j, L_{j+1}, \cdots$. The output of layer L_i will then be passed on to subsequent layers. The forward and backward pass (forward and backward propagation) can thus be computed independently layer by layer, respecting the dependency relation between the layers [11].

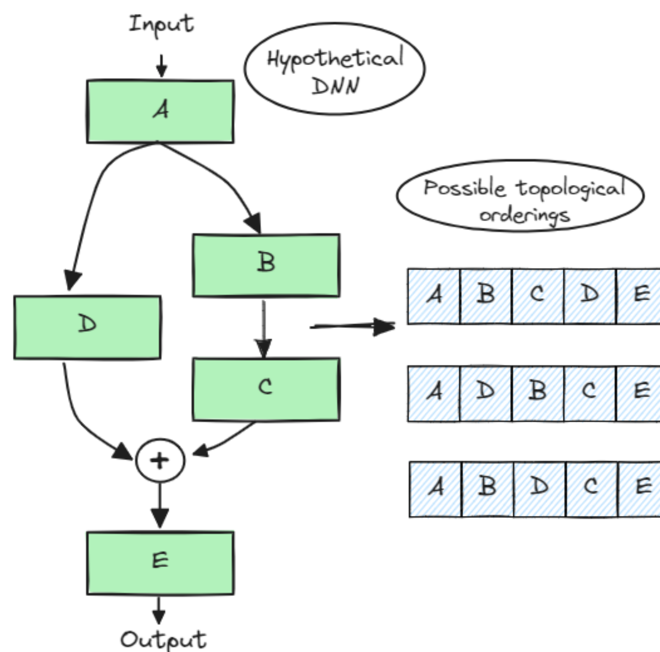

Figure 1.2: Hypothetical DNN example displaying the possible topological orderings. Letters A-E represent layers in a DNN.

In figure 1.2 we can see a hypothetical model and its possible compute orderings (or topological orderings). Layer A has to be first in all orderings. Layers B and D depend on A, and C depends on B. Layer E depends on the output of layers C and D. Note that a layer constitutes a collection of multiple different *kernels*, also in a topologically sorted order. It follows that the graph representation in the figure can be expressed in more detail.

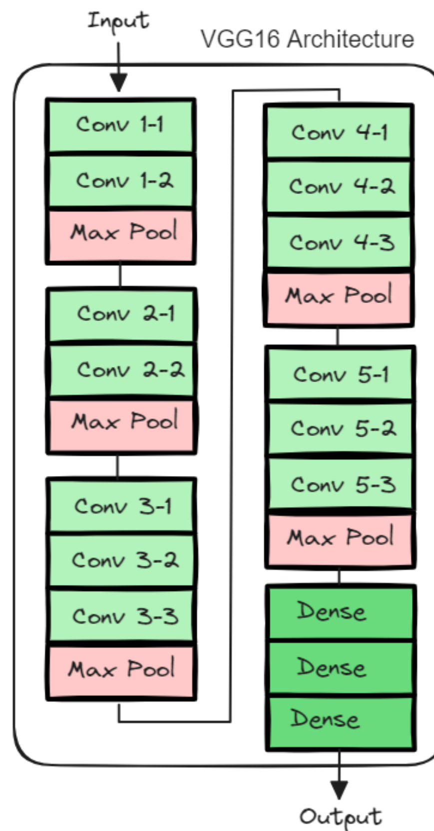

Figure 1.3: VGG16 Architecture with a total of 21 layers, displaying the inherent topological ordering

A good DNN architecture example where this property is portrayed is VGG16, seen in figure 1.3, which consists of 13 *Convolutional* layers, five *Max Pool* layers and three *Dense layers*[15]. Naturally, models can be far more complex, containing up to hundreds of layers and constructing a computational graph that spans thousands of *kernels*.

1.4.2 Execution of Deep Neural Network Models

When, say a forward pass, of a DNN, takes place, in a framework like *pytorch*, there are generally two stages in the execution: *transmission* of the model from physical memory to the device over PCI, denoted T_t, and *execution* of the model with respective input, T_e[4]. For each layer L_i, the transmission and execution is defined as T_{t_i} and T_{e_i}, and total execution time is the sum of transmission and execution time.

$$T_{tot} = \sum_{i=1}^{N} T_{t_i} + T_{e_i} \tag{1.1}$$

Note, equation 1.1 assumes there is already memory allocated on the device for the model, which would otherwise cause extra overhead.

Pipelined Execution

Pipelining execution is a fundamental technique used in computing to enhance performance by parallelizing the processing of instructions. At its core, pipelining divides the execution process into several sequential stages, each capable of operating concurrently. As a result, while one instruction is being executed in one stage of the pipeline, another can be processed in a different stage. This approach significantly increases throughput and maximizes the utilization of computational resources.

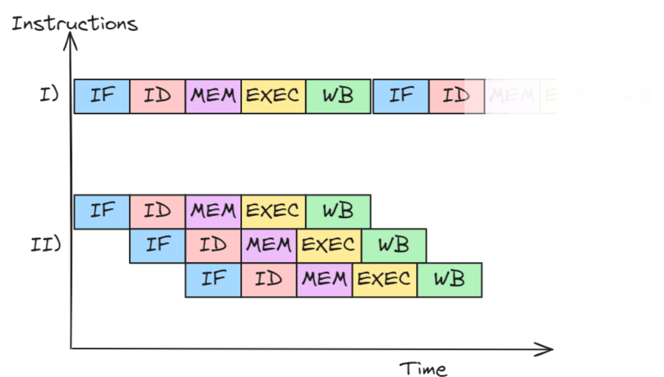

Figure 1.4: Illustration of pipelined execution. Case I shows the sequential execution of an instruction, while case II illustrates the achieved parallelism and speedup achieved by performing the execution process with pipelining.

In figure 1.4, a five-stage pipeline typically found in many processors is illustrated. Stages such as instruction fetch (IF), decode (ID), execute (EXEC), memory access (MEM), and register write-back (WB) can operate simultaneously for different instructions, leading to a more efficient processing system where multiple instructions are handled at various stages of completion, thus accelerating overall execution times.

Deep Neural Network Pipelining

Execution of a DNN can be divided into two stages. The *transmission* of the model parameters over PCIe to the device, and *exection* of the model using some input. In figure 1.5 we can see two cases of execution; linear and pipelined execution.

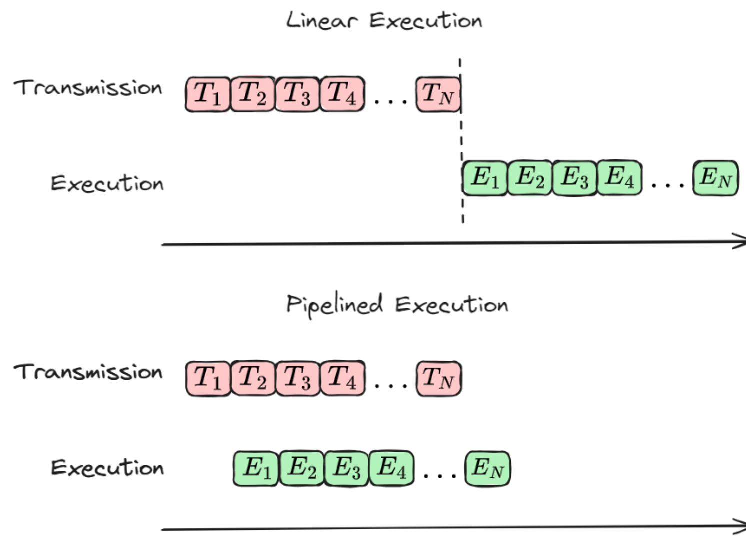

Figure 1.5: Illustration of pipelined execution. Case I shows the sequential execution of an instruction, while case II illustrates the achieved parallelism and speedup achieved by performing the execution process with pipelining.

By pipelining *transmission* and *execution*, the end-to-end request latency can be reduced significantly. Grouping strategies can be applied to find a proper batching technique of layers, as shown in related work. This implies that the choice of several layers transmitted at one time might affect overall execution time and might differ between models. The concept of pipelining execution of deep learning models works in the same way as described for instruction pipelining. The process of executing an inference request for a deep learning model entails (1) transmitting the model *parameters* to the device, (2) performing the forward pass, acquiring a result, and (3) copying the output tensor from the device to physical memory. Since computation is offloaded to the GPU, it requires the model parameters and data to be transferred over the PCIe to device memory. This operation, if done frequently and with large tensors, can become a costly operation. The pipelined execution of the model aims to hide the inflicted latency of transmission and execution by allowing the execution and transmission of the model to take place concurrently. The idea behind the technique is thus to effectively hide execution time inside of transmission time.

One case has been left out, and that is the case of having a model readily available on the device for a request. This case only requires the model parameters to be transferred once and can be executed efficiently thereafter. This case is illustrated in figure 1.6.

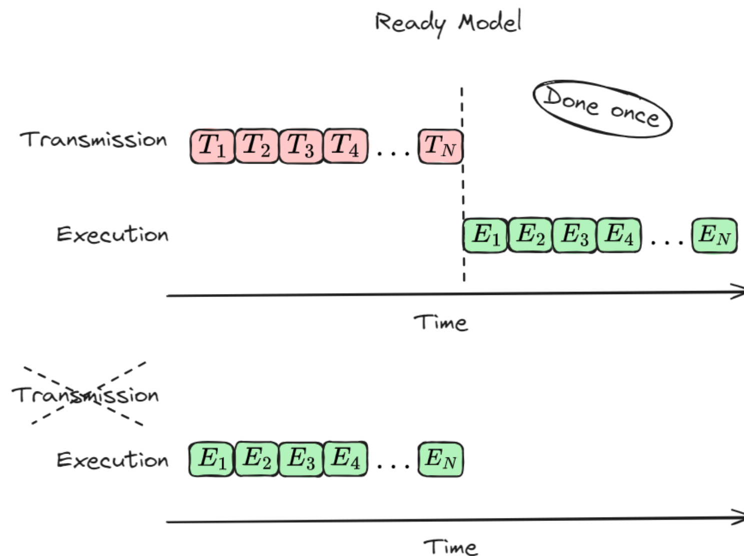

Figure 1.6: Illustration of execution with the model ready in device memory.

The pipelined execution is performed in the same manner for both implementations that we will be evaluating, with only small differences in how the device pointers are acquired.

1.4.3 NVIDIA GPUs

NIVIDA GPUs are complex accelerators which introduce new concepts for a developer to understand when following the *programming model* provided in the CUDA Toolkit.

CUDA Toolkit

The NVIDIA CUDA toolkit provides a development environment for GPU accelerated applications. With it, a developer has access to a range of GPU accelerated libraries for *linear algebra* (cuBLAS), *deep neural networks* (cuDNN), data processing (RAPIDS cuDF) and parallel algorithm *standard library* (Thrust)[13]. Moreover, the *toolkit* gives developers access to the GPU APIs in two different forms of granularity. A *runtime-* and *Driver* API. First we introduce the terminology of entities in the CUDA programming model.

Host and Device memory refer to the system memory (RAM) and the memory that resides on the GPU, respectively.

Kernels are the fundamental unit of execution in CUDA. The *kernel* is a function implemented by the developer, which is executed on the *device*. A kernel is an inherently parallel unit of execution, where the developer can specify the level of parallelism of the kernel. For instance, executing a function N times in parallel by N different threads.

Contexts in CUDA encompass state management, kernel execution, memory management and error handling necessary for a running application.

Modules are dynamically loadable packages of of *device* code and data. The module contains all symbols, functions, global variables and more at module scope. This allows multiple modules to coexist within a single context.

Streams represent a sequence of concurrent operations issued by applications which execute in order. The streams guarantee that commands issued to the stream may execute when all the dependencies of the command are met.

Events are useful synchronization primitives which, put together with *streams*, provide guarantees that all units of work that were submitted to a stream prior the event have finished executing.

CUDA Runtime API is a high level API which performs implicit *context*, *module* and *initialization* management, leading to simpler code. It allows for virtual memory management between *host* and *device*, synchronization of computation and more abstracted operations on the GPU. The runtime API is an abstraction that builds on *low-level* driver API.

CUDA Driver API is the low level library, equipped with fine-grained control mechanisms for *context*, *module*, *stream* and *virtual memory* management and more. The library is a super set of the runtime API and assumes the same capabilities, however, more complex to manage.

Nvidia Multi-Process Service is a tool developed by Nvidia, allowing users to use either spatial or temporal sharing of a single GPU between multiple processes.

Compute Capabilities

CUDA compute capabilities classify NVIDIA GPUs based on their features and performance for parallel computing. Each version of compute capability, such as 7.0, 8.0, and 9.0, corresponds to specific hardware features and performance levels. These capabilities are directly related to different versions of the CUDA toolkit and cuDNN, as each toolkit version supports certain compute capabilities, ensuring compatibility and optimized performance. For instance, newer versions of the CUDA toolkit and cuDNN often introduce features that leverage the advanced capabilities of the latest GPUs, providing better performance and efficiency for complex computational tasks[13]

1.4.4 Pytorch

Fundamental concepts from *pytorch*

A ***Tensor*** in PyTorch is a multi-dimensional array with capabilities optimized for use on varying accelerators, enabling accelerated computing. Tensors are used to encode the inputs and outputs of a model, as well as the model's parameters. They are fundamental to operations in neural networks, facilitating efficient mathematical computations.

PyTorch ***Modules*** are the base class for all neural network modules which includes layers, and often encapsulates parameters, helper methods, and more. Custom networks are defined by subclassing torch.nn.Module and defining forward methods, which take inputs, compute operations, and return outputs.

Hooks are functions that can be registered on a Module or Tensor. They are useful for debugging or understanding the model by allowing for the inspection and modification of outputs and gradients at different points in the network.

The ***Device*** is where Tensor computations are performed, for instance, CPU or GPU.

Dispatcher in PyTorch refers to the mechanism that enables dynamic selection of implementations based on properties of input arguments such as data types and device types. The dispatcher is what allows us to switch between GPU, CPU and others as our choice of accelerator for our tensors.

pybind is a header only library which exposes C++ types in python and vice versa[9]. Used to create python bindings to C++ code.

Pytorch Process Creation

When working with multiple processes in pytorch, you can specify how the created processes will be created and it is then important to know the distinction between the different methods.

The ***fork()*** call creates a new process by duplicating the calling process and works as specified by the equivalent system call. The new process, known as the child process, is an exact copy of the parent process, including the execution state. The parent and child will only duplicate memory when a write is performed to shared memory. This is called *copy-on-write*. This implies, as long as memory is not mutated, duplication will not occur [2].

The ***exec()*** call is used to replace the current process image to instead use some other binary. This allows a calling process to essentially to replace itself with another program [1].

Spawn or ***posix_spawn*** as it is known according to the *POSIX* standard, is a private system call which aims to combine the fork and exec system call into one well defined step of how a process can be created[3] In pytorch, this will entails replacing the process image with a new one, reinitalizing modules and more.

1.4.5 Inter Process Communication

Inter-Process Communication (IPC) is a set of programming interfaces that allow coordination among different program processes which execute concurrently in an operating system[14]. This enables data sharing and task synchronization, despite being separate units of execution.

IPC Mechanisms

Unix systems provide a set of IPC mechanisms that for coordination between multiple related and unrelated processes.

Pipes and Named pipes allow for communication between processes. A *pipe* is a communication channel between two processes that share a common ancestor. *Named pipes* are similar to pipes but are not limited to parent-child processes. They have a name within the file system and can be accessed by unrelated processes.

Message Queues allow processes to share data in the form of messages. Messages take the form of arbitrarily sized blocks. Message queues are provided in System V and POSIX versions.

Shared Memory allows multiple processes to access the same portion of physical memory. It is a fast and efficient form of IPC avoiding the overhead of data copying, allowing direct read and write operations.

Semaphores provide a simple synchronization primitive for processes when performing reads/writes on shared memory. Semaphores come in different implementations on Unix systems, namely System V and POSIX.

Signals are used by the operating system to inform a process of events. Signals are not entirely an IPC mechanism but they can be used to send signals between processes.

1.5 Related Work

1.5.1 Layer Aware Pipelining

In a paper from 2019 introducing GPipe, which is a novel batch-splitting pipeline parallelism algorithm to efficiently train large-scale neural networks by partitioning them across multiple GPUs[8]. The GPipe approach divides a mini-batch of training examples into smaller micro-batches, which are then pipelined through several GPUs, allowing different accelerators to process different micro-batches simultaneously. This design achieves nearly linear speedup with the number of GPUs used, as demonstrated by the significant performance gains in training large models like AmoebaNet and Transformers. However, this design is inherently reliant on multiple GPUs because the partitioning and pipelining strategies are designed to distribute the computational load and memory requirements across several devices. For single GPU scenarios, the benefits of GPipe's parallelism are moot, as the single device must handle the entire model and data processing sequentially, negating the efficiency gains derived from parallel execution and inter-device communication optimizations.

In a paper from 2020[4], an approach to pipelined context switching is introduced, where *training* and *inference* applications can share a GPU with millisecond overhead. By producing metrics of *transmission time (T_i)*, *execution time (E_i)* and *overall delay (D_i)* when executing a given layer L_i, they introduce a novel algorithm for layer grouping, which minimizes the total execution time of a model M. The algorithm achieves optimal grouping given a list of layers for linear and non-linear models, assuming the layered list produced from the non-linear model is topologically sorted. By extending *PyTorch*, and utilizing its memory management component, they introduce a shared cache which multiple Pytorch processes, managed by a server, can access through IPC calls. In conclusion, the provided solution is capable of meeting service-level requirements while increasing GPU utilization for a single GPU workload.

1.5.2 Batch Parallelism and Pipelining

PipeDream [12], presented in a paper from 2019, introduced *inter-batch* pipelining to *intra-batch* parallelism of training workloads of *deep learning* models. Intra-batch parallelism for model *training* implies splitting an iteration of training between multiple workers, leveraging *data-parallelism* and reducing intermediate state results by frequent communication and transmission of data between workers. With *Intra-batch* parallelism, PipeDream parallelized the forward and backward propagation, which is an integral part of *DNN* training. This was accomplished by performing groupings of model layers into mini-batches, which then are partitioned into m micro-batches. This allowed for efficient pipelining of training workloads, pipeline flushing, and, compared to its predecessor GPipe[8], automatic *model partitioning*. PipeDream leverages Pytorch, but is extensible to other frameworks, such as TensorFlow, MXnet[5] and Caffe[10] and runs on Nvidia GPUs utilizing CUDA[13].

The work with PipeDream is, as an extension of GPipe, intended to improve model partitioning over multiple GPUs and not single GPUs. This is outside the scope of the system we seek to evaluate for this thesis.

1.5.3 Microsecond Inference Preemption

Reef [7], presented in a paper from 2022, introduced a strategy for *inference* applications that can perform microsecond preemption of *inference* kernels, also known as real-time kernels. Reef performs kernel fusion, to group high-priority kernels with *best-effort* kernels to maximize throughput and GPU utilization. Reef incurs an overall latency of 2.2%, but increases throughput by about 7x for specific workloads. A con with using Reef is that both the models consisting of *best-effort* and real-time kernels have to be in memory simultaneously for it to work. There is no mechanism in place that allows for switching models in and out efficiently. A positive for Reef is that the system is framework agnostic and does not rely on any Deep Learning System like Pytorch or Tensorflow and can simply execute out of the box if the Deep Learning Libraries have access to the created library which intercepts kernel scheduling calls.

1.5.4 Resource Estimation and Scheduling, Grouping Algorithm

Liquid[6], presented in a paper from 2022, introduces a *Grouping Genetic Algorithm* (GGA) for batch job scheduling in a cluster. Similarly to the grouping strategy performed by PipeSwitch[4], Liquid finds an optimal batch grouping strategy from this GGA. The key thing to note is that the GGA is not used for fine-grained GPU scheduling but for assignment in a cluster based on available resources.

Chapter 2

Approach

This chapter will highlight the process of implementing *pipelined context switching* for Pytorch-2.1.0 and some design choices we made compared to that of Pytorch-1.3.0.

2.1 Method

Our strategy of implementing a solution that can perform context switching of DL models on newer generation GPUs was to look at previous related work. A design we settled on evaluating was the one presented through *PipeSwitch*. We decided on choosing PipeSwitch because of it's intended use case for single GPU execution, and promising performance characteristics as explained in the paper, where transmission overhead was reduced by factors of up to 40x when comparing against Nvidia MPS.

We will be porting PipeSwitch, which is implemented for Pytorch-1.3.0, to Pytorch-2.1.0. To ensure that our implementation is correct, we will be evaluating our implementation using simple image recognition test samples in batches of 8 for three DNN models. For each run we will ensure that our solution does not produce an unexpected result and compare its prediction with that of a model running an unmodified Pytorch 2.1.0. With verification from an unmodified Pytorch library, we can more confidently trust our results and experiments.

When we arrive at an implementation which can perform *pipelined context switching*, we can then perform our experiments and analyze if the transmission overhead of model parameters is decreased or not, as well as how well it performs on newer generations of GPUs. In context, when we have clients that demand that *inference* requests should take no longer than 100ms to perform, we can then compare our results and see if pipelined context switching can help us remain below that bound.

2.1.1 Architecture Overview

In the existing *system* we settled with, four main components deal with incoming connections, workload assignment, execution, and memory management respectively. The pipelined context switching system will at any time consist of three processes: a server and two workers, which are child processes of the server. The server is responsible for *allocating* the shared device memory, spawning worker processes, and exposing the device pointer by transferring it over TCP/IP to the workers. The server will also collect the list of DNN models it will be serving for training and inference requests. Moreover, the server spawns a *controller* which manages the workers, model list, and parameter allocation on the GPU. After all initialization, the server will take care of incoming client connections over TCP/IP and transfer them to the controller, which in turn deals with the request. A simplified illustration of the system can be seen in figure 2.1

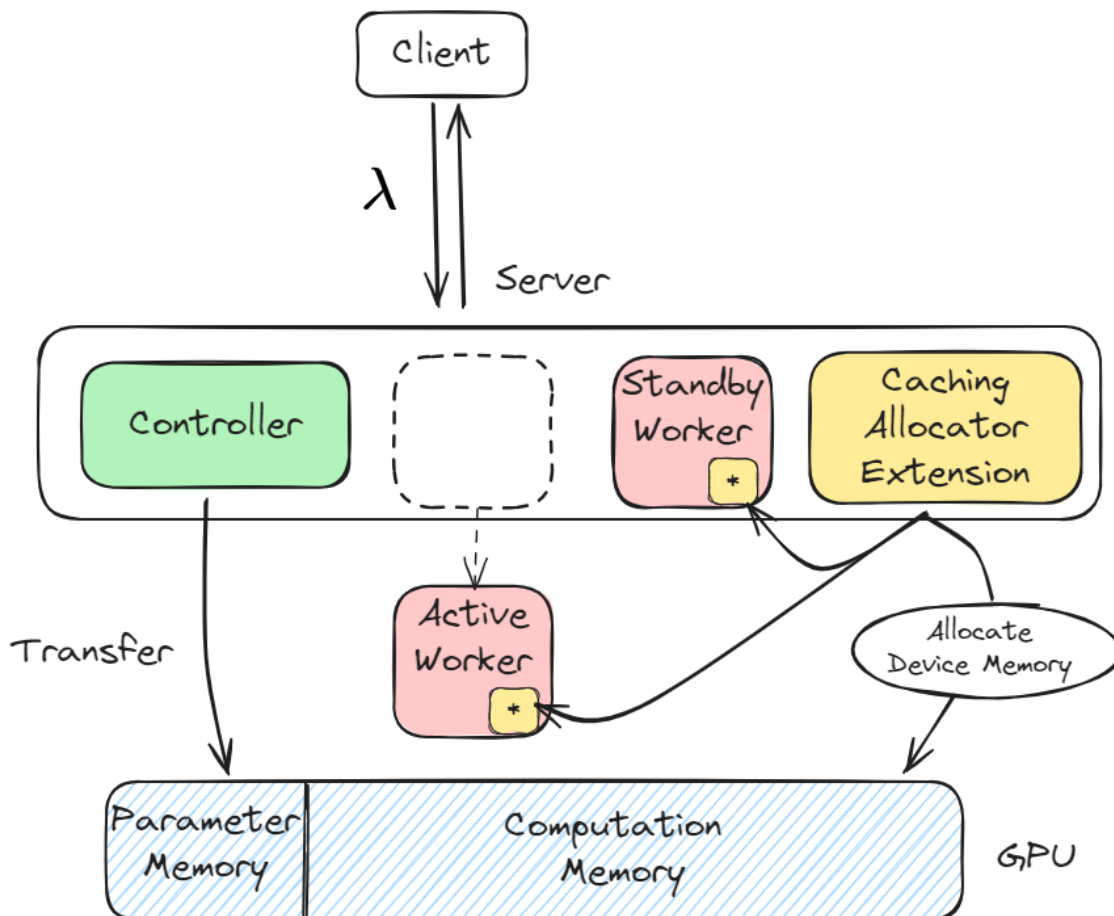

Figure 2.1: Simplified architecture overview. Note that there are a total of 3 separate processes, two workers and a server.

Pytorch has a memory management component that allocates memory for an accelerator, called a *caching allocator*. There is an implementation of a caching allocator, in some form, for each accelerator that Pytorch has chosen to support. In the case of GPUs, Pytorch leverages the *CUDACachingAllocator*, for CPUs the *CPUCachingAllocator*, and more. The allocator is a complex entity that performs allocation, deallocation, event tracing, and graph recording

and can be extended to support more. The caching allocator is not implemented in *python* but instead leverages C++ for performance reasons. The allocator is accessed through a *foreign function interface* (FFI). The CUDACachingAllocator has been extended, adding five operations to support context switching to the Pytorch API. More on the implementation details in section 2.2.2.

The implemented operations are *allocation-*, *sending-* and *receiving-*, *insertion-* and *clearing* a shared cache. As previously mentioned, the server allocates the device memory and sends the device pointer to the workers. The workers in turn await the device pointer and later insert the model parameters into their own shared cache once the pointer is received.

There are two modes of communication that the server uses, TCP/IP when clients are connecting with a request and sending inference data, and IPC when data is passed from the controller to the active worker. The controller leverages pipes when activating a worker, interrupting an active worker, and for data transmission.

2.1.2 Replication of Experiments

To evaluate that our context-switching architecture works as intended, we want to confirm that the experiments used to evaluate it in the first place can be replicated. The experiments presented are for three separate scenarios explained below. What we wish to measure is the *end-to-end latency* of an inference request performed by a client. The experiments, as presented in the results chapter, are what we are using to evaluate the system and confirm that it is performing as we expected.

2.2 Implementation

2.2.1 Port to Pytorch 2.1

As a preface to the implementation details of the port, most of the changes that we have had to introduce in the port have resided in the low-level components of the architecture. We will thus first have to walk through some of the details of the different versions, more specifically, the CUDACachingAllocator.

2.2.2 CUDACachingAllocator in Pytorch 1.3 and 2.1

In Pytorch 2.1, most of the introduced changes do not reside on the *python* side of the Pytorch module, *torch*, but rather on the C++ side. The CUDACachingAllocator has, between 1.3 and 2.1, seen a lot of rework. The source code of the allocator has gone from around 780 LoC to 3500+ LoC. We will first go over how memory allocation and deallocation are performed to get a better understanding of both Pytorch versions. This will make things easier explaining how Pytorch was extended in both versions. Note that the introduced changes described span more than just the `.cpp` files, but also header files.

In figure 2.2 we have a simple overview of the 1.3 CUDACachingAllocator. The Caching Allocator from 1.3. was designed to manage multiple devices, but in a different manner than

that of newer versions. This *device* allocator was called *THCCachingAllocator*. The THC-CachingAllocator is a component inside of the CUDACachingAllocator with the responsibility of managing the memory allocations of all devices.

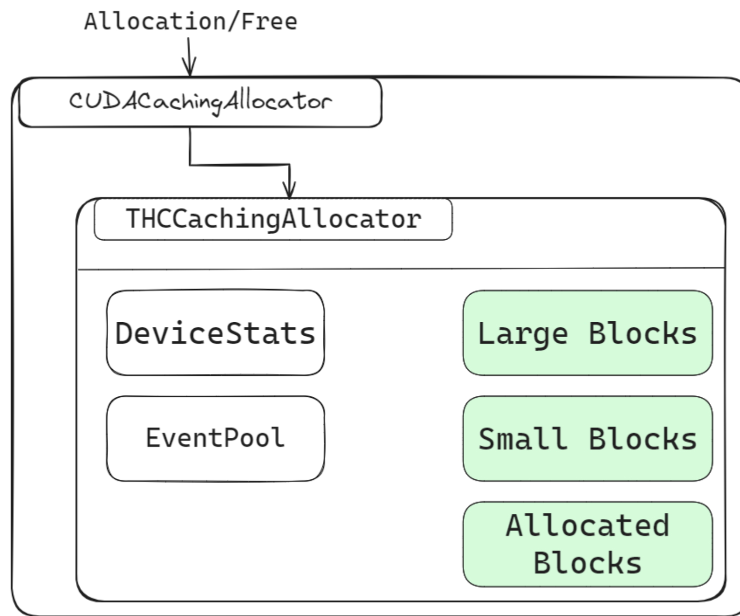

Figure 2.2: Simplified overview of the CUDACachingAllocator from 1.3

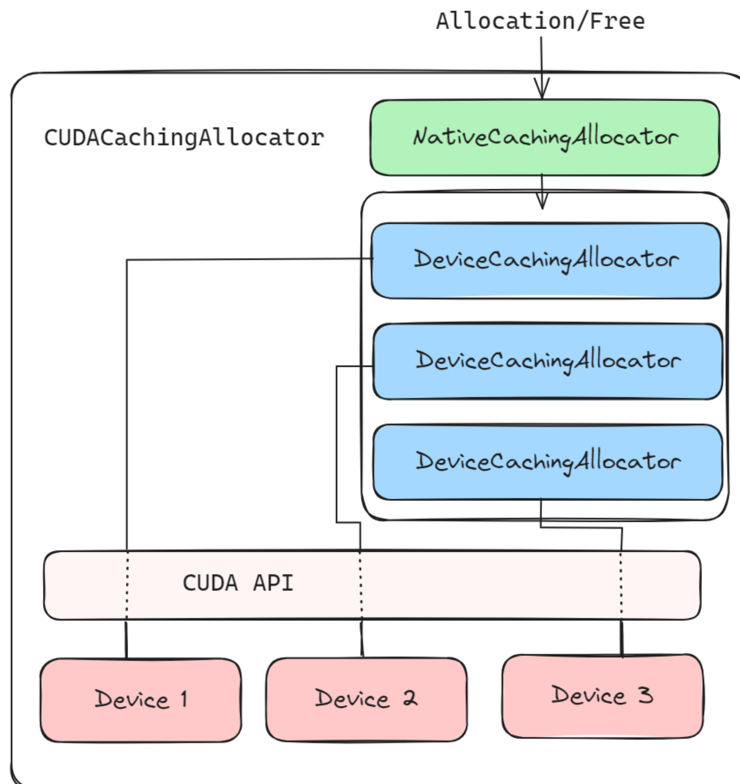

Figure 2.3: Overview of the CUDACachingAllocator from Pytorch 2.1. This is a simplified version, omitting components for initialization.

In figure 2.3 we are presented with an overview of the CUDACachingAllocator in version 2.1. The first thing you can tell from the figure is that there is a proxy module, the *NativeCachingAllocator*, which forwards incoming calls to, now called, *DeviceCachingAllocator*, presented in figure 2.4. The DeviceCachingAllocator is not exposed to outside callers, and if an application interacts with the CUDACachingAllocator, everything is done through the *NativeCachingAllocator*.

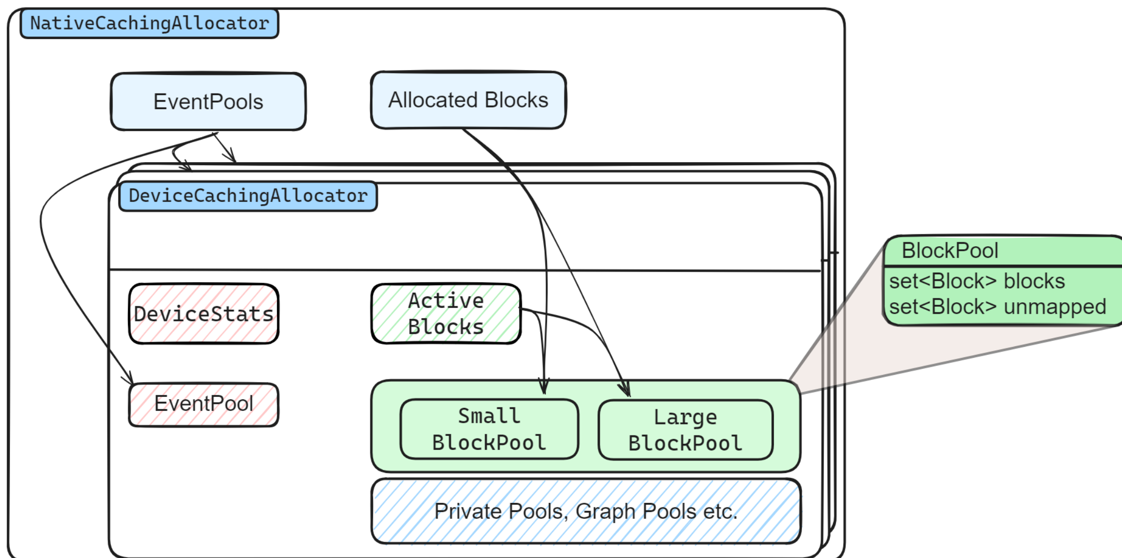

Figure 2.4: The NativeCachingAllocator and DeviceCachingAllocator from pytorch 2.1. Note that this is a simplified illustration.

Memory Allocation and Deallocation

In 1.3, allocations are grouped into two possible sizes. small- and large allocations. A small allocation is an allocation that is smaller than 1MB and a large one is an allocation that is greater or equal to 1MB. Given which group an allocation belongs to, it will reside in either the *small* or *large* BlockPool. A BlockPool is an *Balanced Binary Search Tree* (AVL Tree), known as a *set* in C++, containing all allocated blocks on the device that are either in use or available to be reused. The CUDACachingAllocator also resolves which *stream* that is currently in use by the calling thread. This information is later passed to the THCCachingAllocator. As mentioned earlier, the THCCachingAllocator is responsible for keeping track of all allocations for multiple devices. This can lead to high lock contention and overall performance degradation. *Deallocation* is performed in the same vein as the allocation. The call is forwarded to the THCCachingAllocator, which finds the block of memory a pointer is referencing and deallocates it.

2.1 saw a slight change in this structure. The grouping of allocations is still performed in the same way, however, the BlockPool is now divided into two distinct AVL trees; *blocks* and *unmapped*. *"Blocks"* contains all the active allocations, while *"unmapped"* contains blocks of memory that can be reused.

When allocating memory, the NativeCachingAllocator forwards the *malloc* call by checking the current target *device* of the calling thread and selecting the correct *DeviceCachingAl-*

locator to forward the call to. The DeviceCachingAllocator will fetch the current stream the calling thread is using. Based on the stream and allocation size, it will select a block that is already allocated on the GPU or allocate a new block. A big change with the DeviceCachingAllocator, compared to that of the THCCachingAllocator, is that it is responsible for allocations on one device, and not multiple. Instead, multiple instances of DeviceCachingAllocators exist and can communicate and synchronize on events that might span multiple devices. As illustrated in figure 2.4, the NativeCachingAllocator is responsible for managing all the instances of DeviceCachingAllocators.

Upon a call to *deallocate* memory, the NativeCachingAllocator and DeviceCachingAllocator process the request in the same vein as an allocation by checking the device and stream of a request. The memory of the device pointer can be released and reinserted into the cache, signaling that it can be reused or freed by making a call to the device to free up the memory.

Extension in 1.3

We briefly mentioned that the extension would be introducing new functions that could manage and access the shared cache, used between processes. They are the following:

- `allocate_shared_cache()` (1)
- `send_shared_cache()` (2)
- `recv_shared_cache()` (3)
- `insert_shared_cache_for_parameter()` (4)
- `insert_shared_cache_for_computation()` (5)
- `clear_shared_cache()` (6)

(1) allocates a fixed size block of memory on the device, which is stored in a pointer, called *shared_cache_pointer*, inside each DeviceCachingAllocator. (2) is used to first create a *cudaIpcMemHandle*, which takes the allocated device pointer and makes it accessible between multiple processes, and sends it to a specified port which another process will read from using (3). Upon receiving the memory handle, a process will unpack the memory handle and assign the *shared_cache_pointer* to its contents. The two functions (4) and (5) insert a block each, of a fixed size, into the *large BlockPool*, mapping each block to an address range of the *shared_cache_pointer*. The parameter and computation call will allocate a block of size S_P and S_C respectively, where the total cache size S_{total} is equivalent to $S_{\text{total}} = S_P + S_C$. Lastly, we have the function (6), which removes all the associated blocks from the *large BlockPool*, but ensures that the device pointer itself is not touched. Important to note that the implementation for this was already in place and the baseline of the 2.1 port.

Extension in 2.1

The API introduced above remains the same, with the same function signatures being present. However, we decided to introduce some changes to how the allocated device pointer, obtained when allocating through (1), would be accessed by child processes. What we noticed from 1.3 was a context-switching system that couldn't start up a new process easily if one were to crash due to unexpected failure. We decided that if we replace the communication medium used to transfer the memory handle, we could simplify some things. The server would not have to sequentially initialize the child processes, which was done in 1.3, and we could also

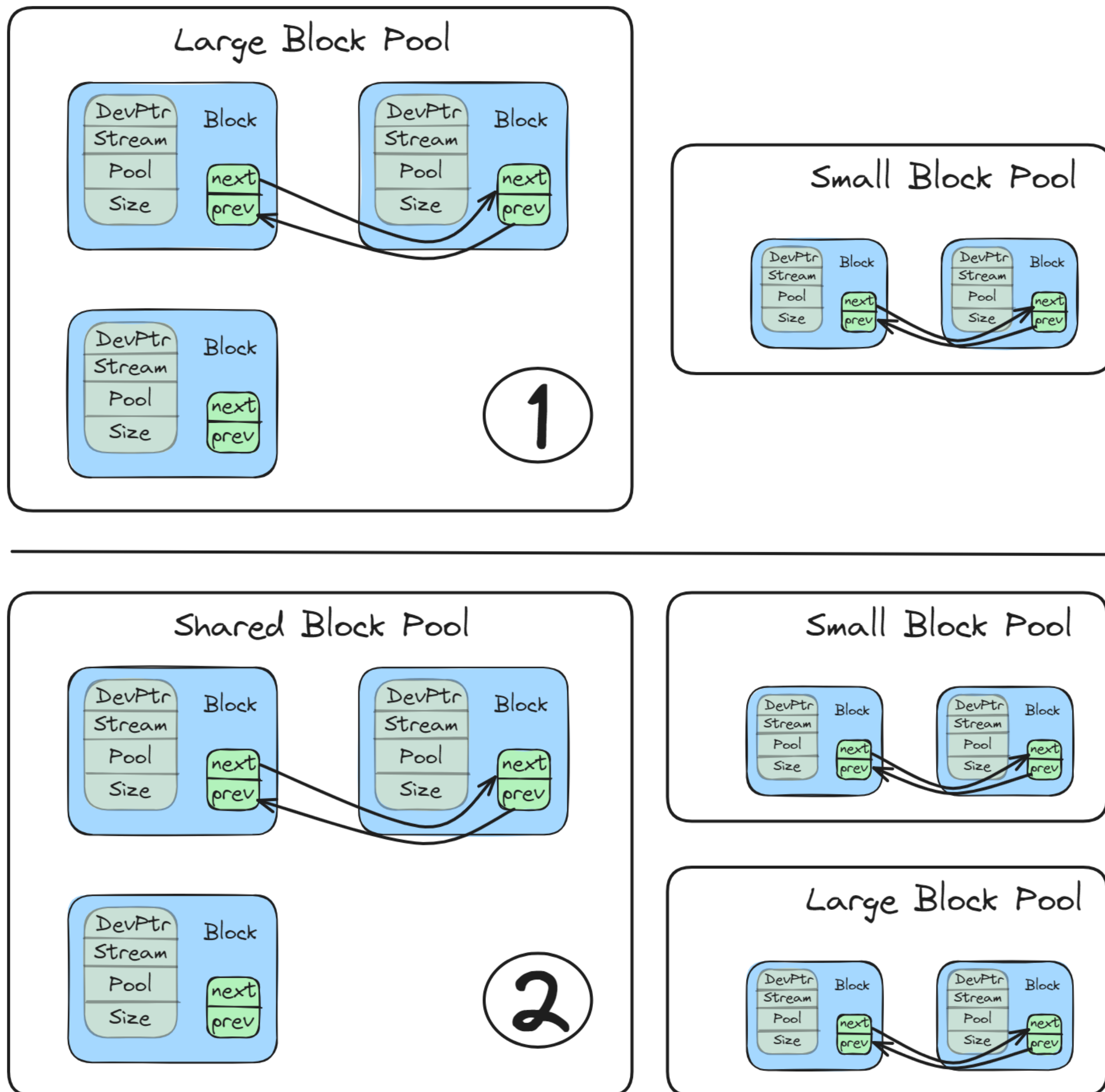

Figure 2.5: (1) is the BlockPool structure for the extension in 1.3 and (2) is for 2.1. All BlockPools are contained inside the DeviceCachingAllocator.

get rid of some poor design choices left on the Python side of the project which struggled with synchronization.

What we instead decided to do was to write the *cudaIpcMemHandle* to a shared memory region, which all processes could access on startup. The server would only have to call (1) to allocate, (2) to create a shared memory segment containing information about the memory handle the children need to access. Execution could then be performed more isolated between the parent and child processes. We could then rest assured that the child processes, once they had started up, would be initialized correctly with minimal interaction.

The biggest difference inside of the CUDACachingAllocator, was to introduce a new type of BlockPool inside the DeviceCachingAllocator, called a *shared pool* (see figure 2.5). Previously, to be able to use the shared memory region inside the worker process, an allocation had to be greater than 1MB and the correct stream had to be allocated correctly. This implies,

that if you are performing allocations that are sub 1MB in size while using the stream, you would allocate more memory inside the small block pool, which is undesirable behavior. To combat this, in 2.1, we introduce a new pool that ensures that all allocations performed, while using a stream associated with the shared pointer, end up in the same place.

As mentioned earlier, clearing the cache using (6) becomes trivial, as we only need to delete all the blocks contained in the shared pool.

2.2.3 Binding to Python

Lastly, to use our introduced changes, we will have to bind our C++ implementation using *pybind*, which is internally used within Pytorch. In figure 2.6, we can see the modified files that constitute the entire patch. As mentioned earlier, the CUDACachingAllocator and related header file are there, however, there is also a file called *Module.cpp*. This file declares all the C/C++ bindings we can access from our Python code. We simply extend this file with our aforementioned functions and ensure we follow pybind documentation.

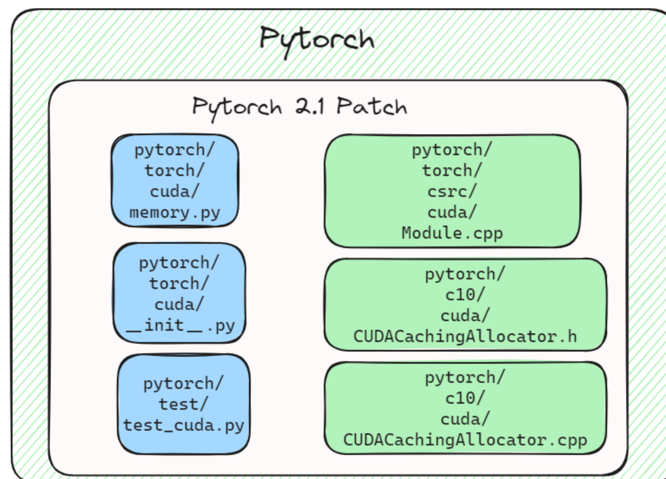

Figure 2.6: All modified files on the Python and C++ side. Pybind is used to bind to the Module on the C++ side.

The `.py` files in figure 2.6, except *test_cuda.py*, are where we declare our Python functions and ensure that they call our C++ bindings. *test_cuda.py* is only used to ensure that we don't break Pytorch itself with our patch.

2.2.4 Workers and Scheduler Memory Sharing

We now have an idea of how the allocation is performed inside the CUDACachingAllocator and how the memory on the device is exposed to all processes that will be executing workloads. However, there is an allocation detail regarding the minimization of transferred data and message passing between processes that is important to understand. When the server initializes all models, the allocation of model parameters is done in a 1GB memory region. The workers will themselves be allocating all the model parameters with that very same offset, resulting in an equivalent mapping for the server to a worker respectively. To demonstrate what we mean by this statement, see figure 2.7

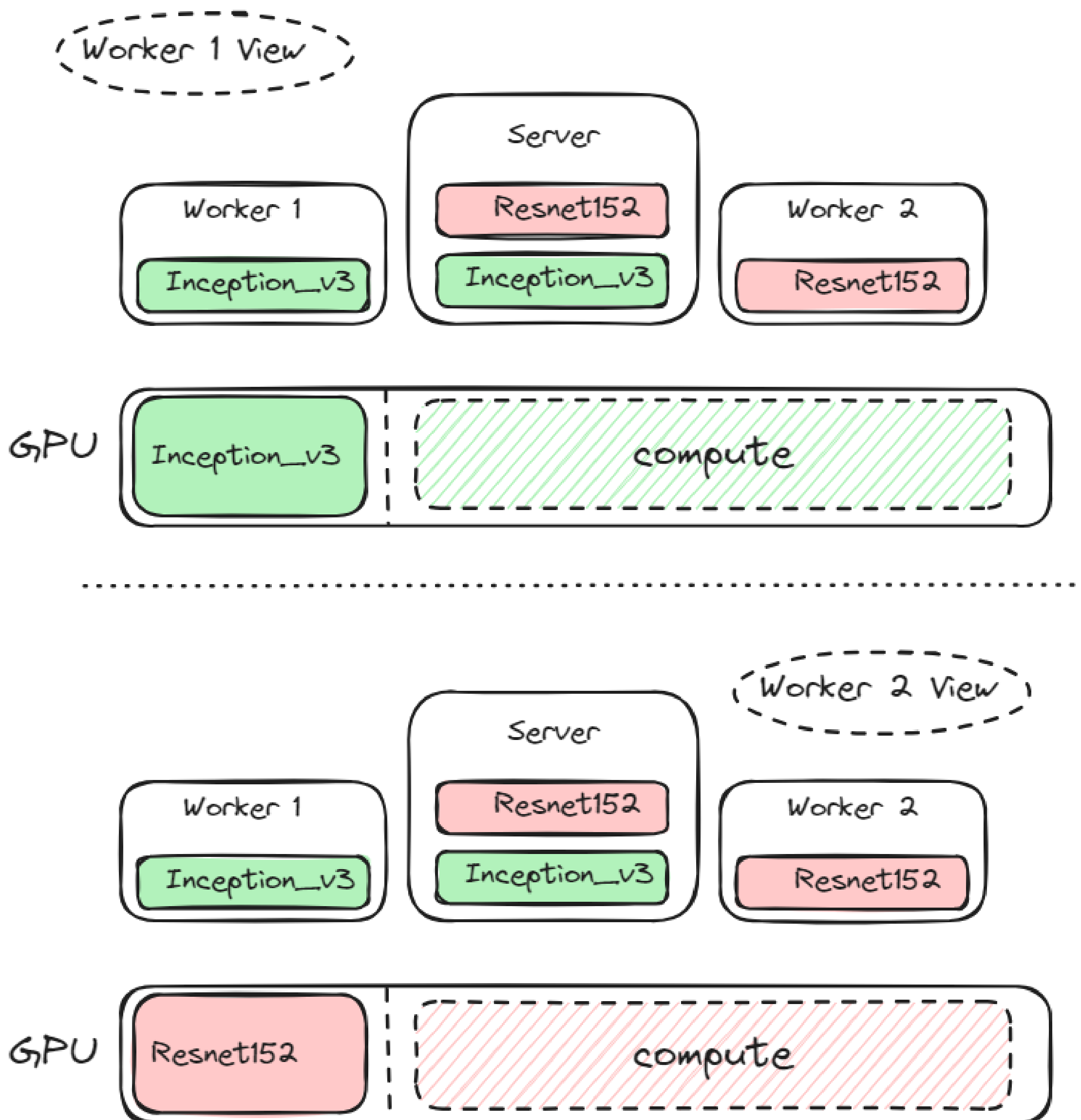

Figure 2.7: View of the memory as seen by workers 1 and 2.

Worker one will be allocating the model parameters of *Inference_v3* while worker two will take care of requests for *Resnet152*. Since the *server* is the one responsible for dealing with transmitting the model parameters to the device, the workers will be unaware of the changes to the state of the device memory. This implies, that for the worker to perform training or respond to an inference request, the server has to only ensure that the right synchronization is taken care of beforehand. Moreover, since the computational graph of the DNN is topologically ordered, the server transmits it in the order in which the worker will be executing it.

2.2.5 Pipelined Execution

Now that we understand how the memory views for both workers and server work respectively as well as how pipelined transmission and execution of deep learning models can be

performed, we need to mention how the execution is performed in practice. Pytorch allows for registration of events on model layers, called *hooks* when a forward pass or backward pass is performed. These events are registrations of functions that will be executed either before or after a forward pass is performed. This behaviour can be specified by the developer.

By inserting forward and backward hooks, workers can synchronize on model transmission with directives from the server using *pipes*. The server can thus ensure that model parameters have been transmitted before any execution takes place, dividing responsibility between server and worker for a forward pass to take place.

Chapter 3

Evaluation

3.1 Experimental Setup

We will be evaluating the aforementioned 1.3 implementation of *context switching* to that of our ported 2.1 version. In this section, we will describe the setup, workloads used, the metrics we use to evaluate the two implementations and showcase some results on different accelerators.

For the experiments, we have access to three **machines**, referred to as Machine 1, 2 and 3. Machine 1 is running *Ubuntu* 22.04.3 LTS x86_64, kernel 5.15.0-91-generic with an Intel Xeon Platinum 8260 (96 cores @ 3.9GHz) CPU. Machine 1 also has two NVIDIA A30 GPUs with 24GB *High Bandwidth Memory* (HBM) 2 and 933 GB/s of GPU memory bandwidth, connected with 8x PCIe Gen4. Machine 2 is running *Ubuntu* 18.04.3 LTS x86_64, kernel 5.15.0-1707-generic, with an Intel Xeon Platinum 8260 (96 cores @ 3.900GHz). Machine 2 is equipped with 3 NVIDIA T4 GPUs with 16GB of memory and 320 GB/s memory bandwidth, connected with 8x PCIe Gen3. Machine 3 is running *Ubuntu* 22.04.03 LTS x86_64, kernel 5.15.05-generic with an Intel Xeon Gold 6354 (72 cores @ 3.600GHz). Machine two has two NVIDIA A40 GPUs with a memory bandwidth of 696 GB/s, PCIe Gen4 interconnect, 48GB GDDR6 GPU memory. The **software** used is mentioned per approach. The Pytorch 1.3 implementation uses PyTorch 1.3, torchvision 0.4.2, scipy 1.3.2 and CUDA 10.1. The 2.1 implementation uses PyTorch 2.1, torchvision 0.16.0, scipy 1.11.4 and CUDA 12.7. We can ensure that all the software is consistent over all machines since we are executing our tests in a *dockerized* environment.

For all our experiments, we have an inference server a client connects to and communicates with using TCP/IP. We want to measure the overall latency of an *inference request* performed using four different strategies, three different **workloads**, on a T4, A30 and A40 GPU and with two separate PyTorch implementations. The different workloads we will be using are *Bert_Base*, *Inception_V3* and *Resnet152*, which are common benchmark models for DL systems. When executing a workload, we will have one worker executing the model in eval-

uation mode, while another is on standby ready for an inference request. When executing our workloads, we are always using a fixed batch size of **8** for our inference requests. Each workload will be executed 100 times, where the first 10 runs will be discarded, as we only want to measure the latency when it has stabilized.

The four different strategies we will use are called:

- ***Ready model***: This is the baseline and best achievable time we can achieve with this experimental setup. The baseline is equivalent to having the model ready to execute server side such that when a client performs an inference request a response can be sent minimizing latency.
- ***Kill and Restart***: This is the worst outcome we can get for an inference request with our setup. Here, the server will preempt a worker process which is training a model and start another tasked to perform an inference request. This implies that the process will have to load the model into host memory, transmit to the GPU and then execute and respond with a result.
- ***Pipelined Switch***: Implies using pipelined execution, pipelining transmission with execution to hide the transmission latency of a model.
- ***Linear Transmission and Execution***: This test case will evaluate the inference latency given that the worker process is not preempted but also does not have the model ready in device memory but has to transmit it from physical memory first without using pipelined execution. This is a good reference point to pipelined context switching.

These strategies are referred to as *Ready Model*, *Kill and Restart*, *Pipelined* and *Linear* in the figures presented in the results section.

3.2 T4 Comparison Results

Before we delve into the acquired results for the two implementations, we have to acknowledge that we lack results for the 1.3 implementation of pipelined execution on the A30 and A40 accelerators. This is one, caused by the CUDA toolkit version we have used for the Pytorch 1.3.0 implementation being incompatible with newer accelerators. Secondly, an error caused by how the *header* files of cuDNN are structured when building Pytorch 1.3.0 from source. Git submodules of Pytorch depend on these header files being ordered in a particular manner for version control. This ordering was changed moving from cuDNN 7.x to 8.x, introducing hard to find bugs when building the project. Since Pytorch 1.3.0 is unsupported as the development has moved on to Pytorch 2.x.x, this remains unfixed. We will thus only compare 1.3.0 and 2.1.0 on the T4 GPU, for which we have data.

3.2.1 1.3.0 and 2.1.0 Comparisons

The relative speedup of the Pytorch 2.1.0 implementation compared to the 1.3.0 implementation, as shown in Table 3.3, provides a clear insight into the performance differences. For *Bert_Base* under the *kill and restart* strategy, the speedup is significant at 2.21x, indicating a more than two-fold increase in efficiency.

In contrast, the *Linear* and *Pipelined* strategies for *Bert_Base* show relative speedups of 0.90x and 0.96x, respectively, suggesting slight regressions. However, the *Ready Model* strategy for *Bert_Base* remains almost on par, with a 0.97x speedup. For *Inception_v3*, the *kill restart* strategy also demonstrates a notable speedup of 2.27x. Both *Linear* and *Pipelined* strategies exhibit modest improvements with speedups of 1.04x and 1.12x, respectively. The *Ready Model* strategy shows a consistent speedup of 1.12x. Lastly, *ResNet152* shows a 1.52x speedup for the *kill restart* strategy. The *Linear*, *Pipelined*, and *Ready Model* strategies all show slight improvements with speedups of around 1.03x to 1.04x.

Overall, the relative speedup analysis underscores the significant advancements in Pytorch 2.1.0 for certain strategies and models but not the pipelined context switching end to end latency which we seek to evaluate.

We can also deduce the produced overhead comparing pipelined context switching and the linear execution to ready model, as seen in table 3.4. Here we can determine that pipelined execution outperforms linear model execution for each test case. For the ResNet152 test case on Pytorch 2.1.0, the overhead is an order of magnitude lower for pipelined execution than that of linear.

Lastly, as seen in figure 3.1 and 3.2, we can see histograms of the execution times, where it is also clear how pipelined execution tends to outperform linear execution.

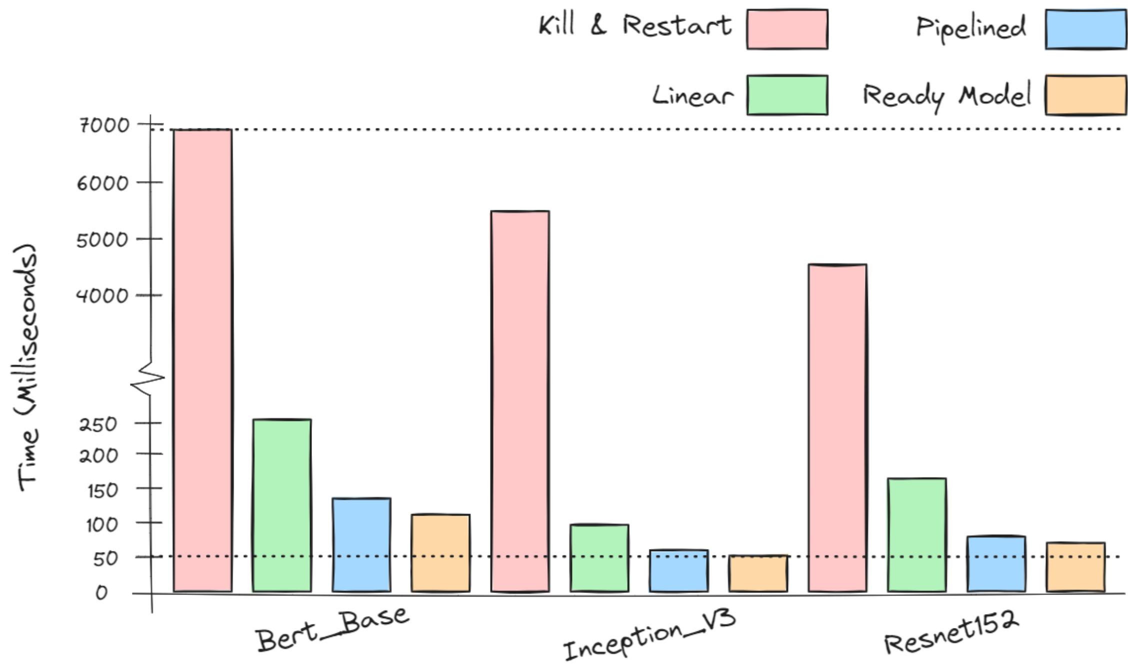

Figure 3.1: Pytorch 1.3.0 implementation executed on a NVIDIA T4 GPU

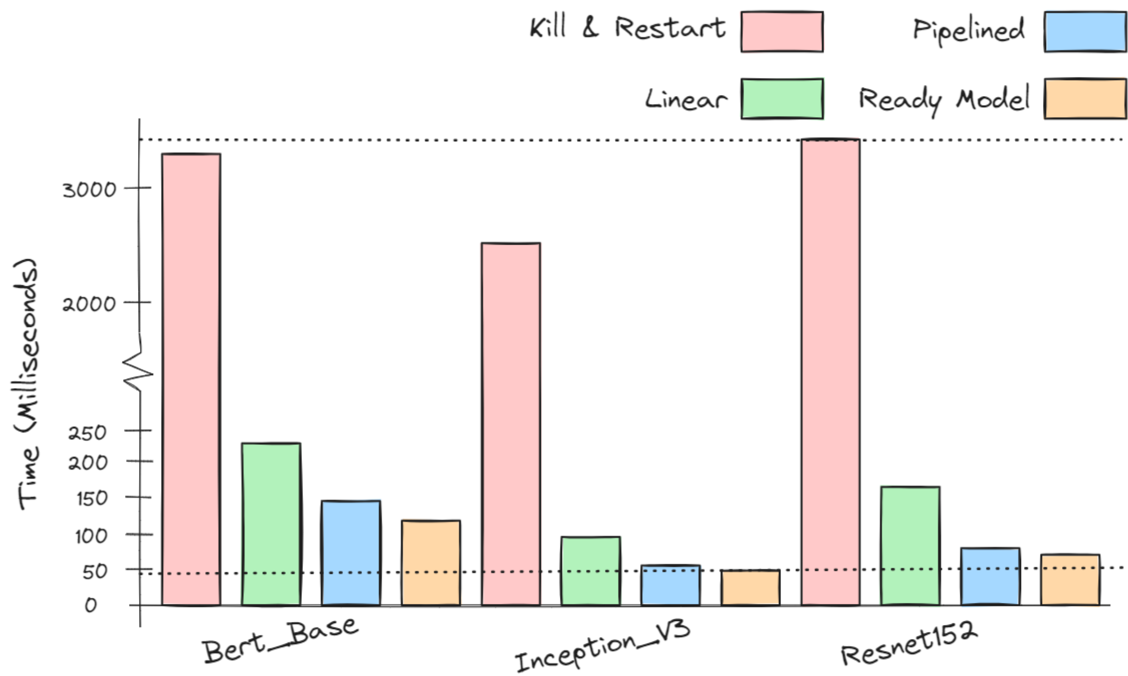

Figure 3.2: Pytorch 2.1.0 implementation executed on a NVIDIA T4 GPU

Table 3.1: Model Performance by Strategy Across All GPU Accelerators for Pytorch 2.1.0

Model	Strategy	Nvidia T4		Nvidia A30		Nvidia A40	
		Value	Unit	Value	Unit	Value	Unit
Bert_Base	Kill Restart	3151.40	ms	3810.79	ms	5901.02	ms
	Linear	226.22	ms	177.64	ms	80.80	ms
	Pipelined	145.65	ms	96.54	ms	73.80	ms
	Ready Model	118.18	ms	62.11	ms	57.06	ms
Inception_v3	Kill Restart	2529.56	ms	2768.57	ms	5098.06	ms
	Linear	95.53	ms	90.53	ms	54.53	ms
	Pipelined	55.47	ms	52.23	ms	41.87	ms
	Ready Model	48.30	ms	35.56	ms	35.12	ms
ResNet152	Kill Restart	3252.21	ms	3812.01	ms	7170.18	ms
	Linear	165.04	ms	77.04	ms	65.77	ms
	Pipelined	79.53	ms	46.21	ms	58.40	ms
	Ready Model	70.63	ms	29.75	ms	37.48	ms

Table 3.2: Model Performance by Strategy Across All GPU Accelerators for Pytorch 1.3

Model	Strategy	Nvidia T4		Nvidia A30		Nvidia A40	
		Value	Unit	Value	Unit	Value	Unit
Bert_Base	Kill Restart	6965.47	ms	-	-	-	-
	Linear	251.04	ms	-	-	-	-
	Pipelined	140.08	ms	-	-	-	-
	Ready Model	115.64	ms	-	-	-	-
Inception_v3	Kill Restart	5736.87	ms	-	-	-	-
	Linear	100.20	ms	-	-	-	-
	Pipelined	62.01	ms	-	-	-	-
	Ready Model	53.92	ms	-	-	-	-
ResNet152	Kill Restart	4932.86	ms	-	-	-	-
	Linear	170.21	ms	-	-	-	-
	Pipelined	82.43	ms	-	-	-	-
	Ready Model	73.23	ms	-	-	-	-

Table 3.3: Relative Speedup of 2.1.0 implementation comparing to 1.3.0 implementation on the T4 accelerator.

Model	Strategy	Relative Speedup
Bert_Base	Kill Restart	2.21x
	Linear	0.90x
	Pipelined	0.96
	Ready Model	0.97x
Inception_v3	Kill Restart	2.27x
	Linear	1.04x
	Pipelined	1.12x
	Ready Model	1.12x
ResNet152	Kill Restart	1.52x
	Linear	1.03x
	Pipelined	1.04x
	Ready Model	1.04x

Table 3.4: Overhead in *milliseconds* of Linear and Pipelined Execution Compared to Ready Model for Pytorch 1.3.0 and 2.1.0 on the T4 Accelearator

Model	Strategy	Overhead (ms)	
		Linear	**Pipelined**
Pytorch 1.3.0			
Bert_Base		135.40	24.44
Inception_v3		46.28	8.09
ResNet152		96.98	9.2
Pytorch 2.1.0			
Bert_Base		108.04	27.47
Inception_v3		47.23	7.17
ResNet152		96.98	9.2

3.3 T4, A30 and A40 Results for Pytorch 2.1.0

We are going to extrapolate on the data as seen in Table 3.1 and focus on the columns specifying the execution times on A30 and A40. Moreover, a histogram for the A30 and A40 is presented in figures 3.3 and 3.4.

Table 3.5 provides the overhead values for pipelined execution compared to the ready model across the different GPU accelerators. For the Bert_Base model, the overhead for linear execution on the Nvidia T4 is 108.04 ms, on the Nvidia A30 is 115.53 ms, and on the Nvidia A40 is 23.74 ms. For pipelined execution of the Bert_Base model, the overhead on the Nvidia T4 is 27.47 ms, on the Nvidia A30 is 34.43 ms, and on the Nvidia A40 is 16.74 ms. The Inception_v3 model shows an overhead for linear execution of 47.23 ms on the Nvidia T4, 54.97 ms on the Nvidia A30, and 19.41 ms on the Nvidia A40. For pipelined execution of the Inception_v3 model, the overhead is 7.17 ms on the Nvidia T4, 16.67 ms on the Nvidia A30, and 6.75 ms on the Nvidia A40. The ResNet152 model exhibits an overhead for linear execution of 94.41 ms on the Nvidia T4, 136.44 ms on the Nvidia A30, and 28.29 ms on the Nvidia A40. For the pipelined execution of the ResNet152 model, the overhead on the Nvidia T4 is 8.90 ms, on the Nvidia A30 is 16.46 ms, and on the Nvidia A40 is 20.92 ms.

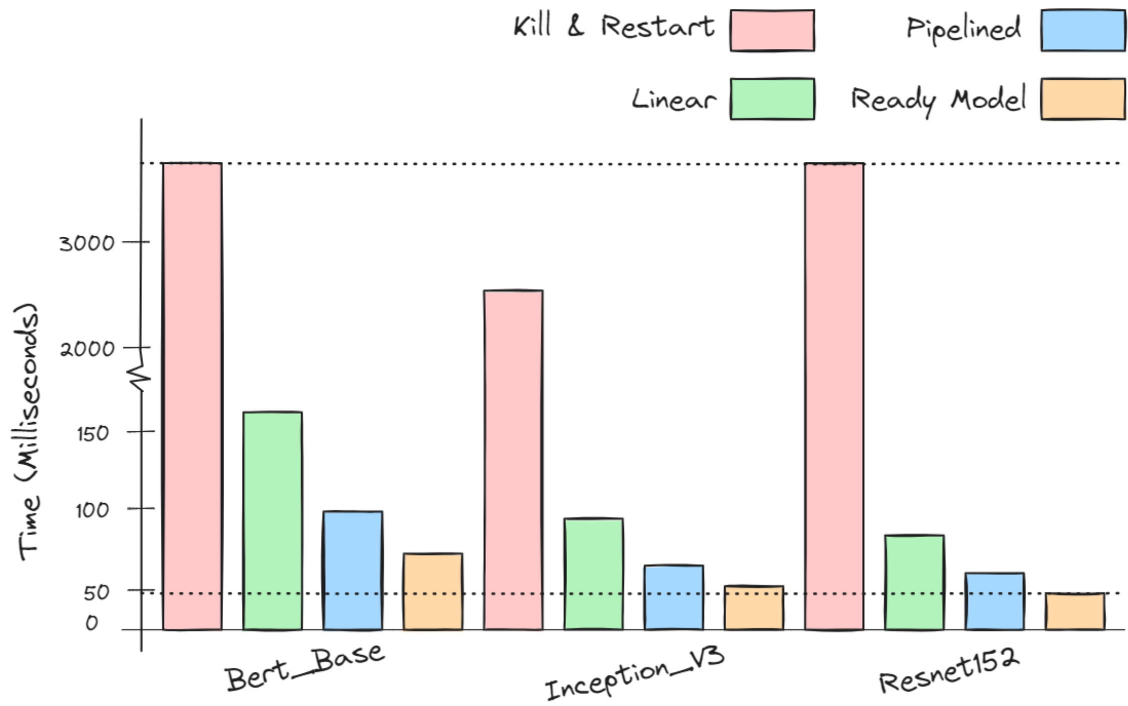

Figure 3.3: Pytorch 2.1 implementation executed on a NVIDIA A30 GPU

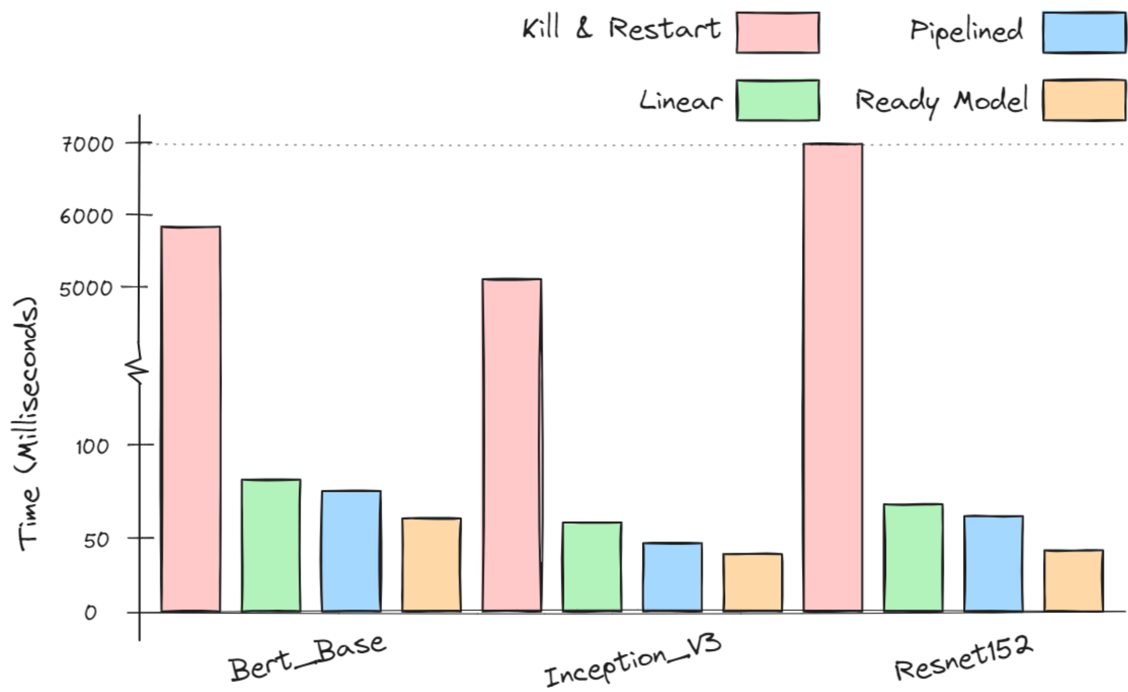

Figure 3.4: Pytorch 2.1 implementation executed on a NVIDIA A40 GPU

Table 3.5: Overhead of Pipelined Context Switch and Linear Execution Compared to Ready Model for Pytorch 2.1.0

Model	Strategy	Nvidia T4 (ms)	Nvidia A30 (ms)	Nvidia A40 (ms)
Bert_Base	Linear	108.04	115.53	23.74
	Pipelined	27.47	34.43	16.74
Inception_v3	Linear	47.23	54.97	19.41
	Pipelined	7.17	16.67	6.75
ResNet152	Linear	94.41	47.29	27.92
	Pipelined	8.90	16.46	20.92

Table 3.6: Table Derived from Table 3.5, Showcasing the Overhead Reduction from Pipelined Execution.

Model	Nvidia T4	Nvidia A30	Nvidia A40
Bert_Base	$\approx 3.93x$	$\approx 3.36x$	$\approx 1.42x$
Inception_v3	$\approx 6.59x$	$\approx 3.30x$	$\approx 2.88x$
ResNet152	$\approx 10.61x$	$\approx 2.87x$	$\approx 1.35x$

Chapter 4

Discussion and Conclusion

This chapter will analyze and reflect on the achieved results from the previous chapter. We will discuss the evaluation of the two implementations on the T4 accelerator as well as evaluate the overhead reduction from the pipelined and linear case. We will then consider restate the problem, the novel contribution from this thesis and how it can be extended in future work.

4.1 Evaluation of Results

4.1.1 Pytorch 1.3.0 and 2.1.0 implementations

As shown in the previous section in table 3.3, derived from table 3.1 and 3.1, there were but slight differences in performance. Most notably, we can see that there has been significant improvement for the kill and restart scenario between 1.3.0 and 2.1.0, with an improvement of up to $2.27x$. However, it becomes non-essential to the reader as it is not a strategy we are trying to evaluate, but use as an upper bound for our experiments. For the other strategies, *linear* and *pipelined* we see but slight improvements or regressions.

The largest regression we saw was $0.90x$ relative speedup of 2.1.0 using the *bert base* model. It is hard to know exactly why we are seeing this decrease in performance since it can have anything to do with a different model implementation that is intended for newer accelerators or changes in the Pytorch architecture that is slowing down execution. We can argue the same way about the seen improvements as the same parameters can account for an improvement in execution time as well.

Looking at table 3.4, we can see the overheads produced by the two different approaches and how much the overhead can be reduced given pipelined context switching. The biggest improvement is, as mentioned earlier, an improvement on the Resnet152 test case, were both implementations sport a near 10x reduction in overhead compared to ready model.

4.1.2 Pipelined Context Switching and Linear Transmission

Overall, we are seeing a trend of pipelined context switching outperforming linear execution. In table 3.6, each cell is equivalent to the fraction of $\frac{\textit{Linear Overhead}}{\text{Pipelined Overhead}}$, which gives us an approximation of how much lower the overhead is comparatively. We can see the same thing here, as mentioned earlier, that for ResNet152 we have an order of magnitude improvement, and still over 8x improvement in overhead reduction for the A30. The A40 is where we see the smallest average reduction in overhead.

The numbers displayed for the 2.1.0 implementation in Tables 3.5 and 3.6 underscore a transitive effect that warrants further analysis. As outlined in the introduction, our primary objective was to identify a system capable of enhancing single GPU utilization. The data presented here suggests that our system demonstrates a marked increase in the time allocated to executing inference requests as opposed to transmitting model parameters. This observation indicates a higher degree of GPU utilization, which is critical for optimizing computational efficiency.

By extrapolating the current findings to a scenario where the number of requests is orders of magnitude higher than those documented in our experiment, it becomes evident that the throughput of *pipelined* inference requests would substantially exceed that of the *linear* model swapping approach. This increased throughput is attributed to the reduction in overhead associated with model parameter transmission, allowing the GPU to devote more resources to execution of workloads instead of remaining idle. Consequently, the pipeline method not only enhances performance but also ensures a more efficient utilization of GPU resources, aligning with our initial hypothesis and objectives. This underscores the potential of the pipelined context switching implementation to improve the GPU utilization when models are frequently swapped in and out of memory.

4.1.3 Pipelined Execution to Meet SLO Requirements

Observing the overhead results, as mentioned in the previous section in table 3.5, a system that needs to swap models in and out of device memory frequently due to resource limitations or otherwise, can benefit greatly in reducing the overall incurred overhead from performing linear transmission. With an order of magnitude reduction, pipelined context switching can greatly benefit clients who suffer from under utilized GPUs. We have seen similar needs in customers of Huawei who provide cloud services with inference as a service which has strict requirements on overall inference request delay.

4.2 Problem Summary

The work presented in this thesis addresses the challenge of optimizing GPU resource utilization for deep learning models. As the adoption of deep learning models increases across various industries, the demand for efficient computational resource management grows. Training state-of-the-art models on GPU clusters is expensive and resource-intensive. This thesis aims to evaluate an existing system for pipelined context switching, which is designed to enhance

the efficiency of single GPU utilization. The goal is to enable multiple models that may not fit in memory concurrently to share GPU resources while minimizing the overhead associated with model switching. The system must meet service level objectives (SLOs) for client inference requests while optimizing resource utilization.

4.2.1 Research Question One

Can pipelined execution of *Deep Neural Networks* still be viable for more recent (Ampere) Nvidia GPUs with regards to existing implementations? Moreover, do we see any improvements in GPU utilization for these devices? For the A30 and A40 accelerators we find that pipelined context switching is viable with significant improvements to the delay caused by model transmission in the non-pipelined case. As discussed in section 4.1.2, we also note that the throughput of inference requests must have increased, and thus the time spend executing requests must have increased as well. This transitive effect supports similar results as expressed in previous work done in PipeSwitch[4], where they found that single GPU utilization was increased by the use of pipelined context switching.

4.2.2 Research Question Two

Can pipelined context switching be used to help solutions reach service level objectives? We strongly believe clients who have a platform in which models are frequently being swapped in and out of memory or want to share the same resources used for training with inference, then pipelined context switching can be of great value to meet *service level objectives*. We think that the aforementioned results speak for great performance improvements compared to transmission of model parameters in a non-pipelined fashion.

4.3 Contribution

The primary contribution of this research is an in-depth analysis of how a pipelined context switching system performs on newer generations of Nvidia GPUs, specifically focusing on Pytorch implementations. The study involves porting the existing pipelined context switching system from Pytorch 1.3.0 to Pytorch 2.1.0 and evaluating its performance across different GPU models, including Nvidia T4, A30, and A40. The analysis provides insights into the effectiveness of pipelined execution in reducing model swapping overhead while meeting the *service level objectivces*. By the end of this project, readers will gain a comprehensive understanding of how pipelined context switching can be utilized to optimize GPU resource usage for deep learning models on modern hardware.

The numbers presented in the aforementioned tables 3.5 and 3.6 affirm that a system performing pipelined context switching decrease switching overhead which can benefit a general system that adheres to strict SLO requirements and wants to allow for multiple models to share a single GPU.

4.4 Future Work

In this section we will discuss aspects that we deem valuable to explore in further work on pipelined context switching and challenges to consider for future implementations.

4.4.1 Preempting Training Jobs

The question for pipelined context switching is how it deals with state management. As of now, we haven't put in the necessary steps to save the state of a model that is being trained properly. As that is a hard engineering challenge already, the question is what benefit would one get from transmitting model parameters for a training job that will be preempted within a short time span. Is it maybe better to keep said model on a different device with sufficient resources or not? Or maybe the incoming inference requests are sparse and thus capturing parameter state might not have to be done as frequently. This is up to others to investigate, but could be of value when implementing more sophisticated solutions that incorporate pipelined context switching.

4.4.2 Framework Agnostic Implementation

An interesting extension to the work presented in this thesis should entail framework agnostic pipelined execution, in other words an implementation unaware of Pytorch or Tensorflow specific details. There is a trade off to consider with new complexities going outside of an established framework, but, if done successfully, could prove to be valuable. This is already done in different manners, for instance as presented in Reef where they perform framework agnostic kernel throttling[7].

Bibliography

[1] *exec(3) Linux Programmer's Manual*, August 2019.

[2] *fork(2) Linux Programmer's Manual*, June 2020.

[3] *posix_spawn(3) Linux Programmer's Manual*, June 2023.

[4] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514. USENIX Association, November 2020.

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[6] Rong Gu, Yuquan Chen, Shuai Liu, Haipeng Dai, Guihai Chen, Kai Zhang, Yang Che, and Yihua Huang. Liquid: Intelligent resource estimation and network-efficient scheduling for deep learning jobs on distributed gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):2808–2820, 2022.

[7] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[8] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. *GPipe: efficient training of giant neural networks using pipeline parallelism*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[9] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 — seamless operability between c++11 and python, 2017. https://github.com/pybind/pybind11.

[10] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[11] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[12] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Gregory Ganger, Phillip Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. pages 1–15, 10 2019.

[13] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[14] David L Presotto and Dennis M Ritchie. Interprocess communication in the ninth edition Unix system. *Software: Practice and Experience*, 20(S1):S3–S17, 1990.

[15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[16] Backlinko Team. ChatGPT / OpenAI Statistics: How Many People Use ChatGPT?

[17] Gokul Yenduri, Ramalingam M, Chemmalar Selvi G, Supriya Y, Gautam Srivastava, Praveen Kumar Reddy Maddikunta, Deepti Raj G, Rutvij H Jhaveri, Prabadevi B, Weizheng Wang, Athanasios V. Vasilakos, and Thippa Reddy Gadekallu. Generative pre-trained transformer: A comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions, 2023.

EXAMENSARBETE Pipelined Context Switching for Deep Learning Models
STUDENT Fredrik Horn af Åminne Dannert
HANDLEDARE Jonas Skeppstedt
EXAMINATOR Michael Doggett

Pipelining av Djupinlärningsmodeller på Grafikprocessorer

POPULÄRVETENSKAPLIG SAMMANFATTNING **Fredrik Horn af Åminne Dannert**

Pipelining är en metod för att parallelisera delmoment av en process. Detta arbete undersöker hur *pipelining* kan underlätta att dela på senare generationens grafikprocessorer (GPU:er) mellan många modeller och samtidigt möta satta prestandakrav.

I det många idag kallar för en AI-revolution har de acceleratorerna som bygger upp den nödvändiga *infrastrukturen* för träning och körning av sofistikerade maskininlärningsmodeller blivit väldigt eftertraktad. När infrastrukturen, som i stor bredd består av grafikprocessorer, är kostsam gäller det att kunna utnyttja dem effektivt när modeller tränas eller används som service, även kallat *inferens*.

Först och främst, varför vill man använda GPU:er för maskininlärningsmodeller? Det GPU:er först var speciellt framtagna för var grafikrendrering, bestående av en många operationer som involverar matrisberäkningar. Då maskininlärning byggs upp av samma operationer så är GPU:er optimala, och mycket effektivare än en CPU, att träna dessa modeller på.

I mitt examensarbete har jag utforskat hur en metod, kallad pipelining, används för att minska tiden som krävs innan en modell är redo att brukas på en GPU, för antingen träning eller inferens. Då den tagna tiden minskar för att ladda in modellen i GPU:ers minne, så kan allt fler modeller dela på en GPU samtidigt som de möter satta prestandakrav. Prestandakraven är vanligen en maximal tolerans för *responstiden* för en service. Exempelvis tiden tagen för en modell som gör bildigenkänning eller en *Large Language Model* (LLM) som svarar på en förfrågan åt en användare.

Pipelining innebär att man skapar överlapp mellan överföring och exekvering av en modell. I figur 1 kan vi se hur de lager som bygger upp en modell skickas i grupperingar och exekveras med överlapp. På så vis kan man gömma en del av exekveringstiden i överföringen och skicka ett svar snabbare till användaren.

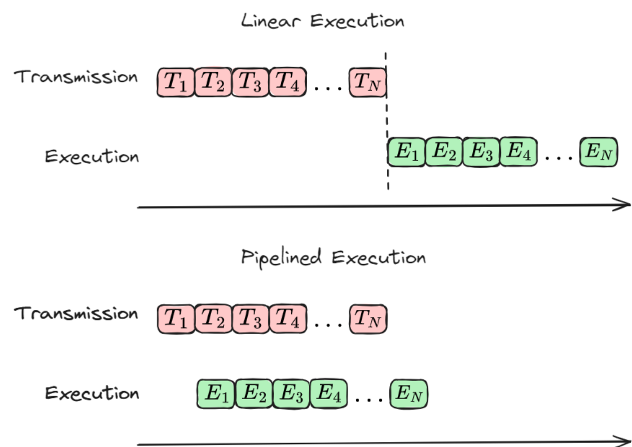

Figure 1: Överföring gjort på vanligt sätt och med pipelining.

Resultaten vi hittade för *Tesla* och *Ampere* generationen av Nvidia GPU:er, visar att pipelining kan hjälpa till att minimera överföringskostnaden och göra det möjligt att enklare möte prestandakrav. Som konsekvens kan flera modeller enklare dela på samma GPU resurs så att den kan utnyttjas mer effektivt.