공학박사학위논문

A Deep Learning Optimization Framework for Versatile GPU Workloads

다양한 종류의 연산을 지원하는 GPU 기반 딥 러닝 최적화 프레임워크 개발

2021 년 8 월

서울대학교 대학원

전기·컴퓨터 공학부

정 우 근

A Deep Learning Optimization Framework for Versatile GPU Workloads

다양한 종류의 연산을 지원하는 GPU 기반 딥 러닝 최적화 프레임워크 개발

지도교수 이 재 진

이 논문을 공학박사학위논문으로 제출함

2021 년 4 월

서울대학교 대학원

전기·컴퓨터 공학부

정 우 근

정우근의 공학박사 학위논문을 인준함

2021 년 7 월

위 원 장	김 진 수
부위원장	이 재 진
위 원	문 수 묵
위 원	정 창 희
위 원	조 형 민

Abstract

Deep Learning (DL) frameworks, such as TensorFlow and PyTorch, are widely used in various fields of applications. However, there are two limitations to these frameworks. The first limitation is that these frameworks heavily rely on the NVIDIA cuDNN for performance. Using cuDNN does not always give the best performance. One reason is that it is hard to handle every case of versatile DNN models and GPU architectures with a library that has a fixed implementation. Another reason is that cuDNN lacks kernel fusion functionality that gives a lot of chances to improve performance. The other limitation is that these frameworks do not support the training of an out-of-GPU-memory-scale model, which is essential for state-of-the-art DL techniques in the field of natural language processing.

In this thesis, we propose two performance-model-based deep learning optimization frameworks. The first is DeepCuts, a DL optimization framework for versatile workloads on a single GPU. It considers both the kernel implementation parameters and GPU architectures. It analyzes the DL workload, groups multiple DL operations into a single GPU kernel, and generates optimized GPU kernels when considering the kernel implementation parameters and GPU architecture parameters. The evaluation results under various DL workloads for inference and training indicate that DeepCuts outperforms cuDNN/cuBLAS-based implementations and the state-of-the-art DL optimization frameworks, such as TVM, TensorFlow XLA, and TensorRT.

In addition, we propose BigCuts: a DL optimization framework for out-of-GPU-memory scale model training on a GPU cluster. BigCuts takes the

information of the target model to train and the target cluster system to use and finds the best way of cluster utilization using the performance model. BigCuts also considers using a storage system to train a massively scaled model whose parameters cannot be stored in CPU memory. BigCuts successfully find the best way of cluster utilization for a versatile type of models and hardware systems, unlike previous approaches such as Megatron and DeepSpeed, which lack such capability.

Keywords: GPU, Deep Learning, Code Generation, Parallelization, Cluster, Performance Modeling

Student Number: 2012-20860

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep learning (DL) has been rapidly developed and used in a range of domains, including image generation[1], automation of gameplay[2], speech recognition[3], and natural language processing[4]. The great success of DL mainly comes from the advances in hardware systems, and the adaption of software frameworks for the systems. During the last decade, NVIDIA GPUs, and GPU-based clusters have become the *de facto* standard for running deep learning applications. Widely used deep learning frameworks, such as PyTorch[5] and TensorFlow[6] have adapted to these systems, and provide easy-to-use programming interfaces for them.

However, the rapid advances of the DL techniques in recent years have resulted in certain changes. There are two major trends for the current state-of-the-art advances of deep learning techniques. The first trend is the *versatility*. After the great success of the DL technique in the field of image classification[7], the advances of the DL techniques were mainly focused on CNN-based image processing that fully utilizes the computational power of the GPU. However,

in recent years, versatile non-CNN models, including RNNs and transformers, and non-conventional types of workloads (e.g., edge-targeted lightweight model inferences) have been proposed and widely used.

The other trend is the *rapid increase in parameter size*. After the great success of huge transformer models such as BERT[4] and GPT[8], the size of the models for natural language processing has rapidly been increasing. For instance, although widely-used models developed before 2018[9][10][4] have less than 1B parameters, recently-developed transformer models[8][11] have more than 100B parameters.

Software frameworks have failed to adapt to these advances. Almost every widely used DL framework supports GPU acceleration through cuDNN[12], the state-of-the-art DL primitive library. However, cuDNN-based DL frameworks do not always achieve the best performance, because of the versatility of DL workloads. Although cuDNN is mainly optimized for CNN inference and training with a large batch of inputs, emerging DL applications may require accelerating different computations. For instance, various types of non-convolution DL operations (*e.g.*, LSTM, GRU, embedding, etc.) are rapidly developed and widely used today. cuDNN is not fully optimized or lacks support for such operations. The input batch size also matters. Because the batch size is determined by the environment where the DNN model is used, it has difficulty handling to handle all batch sizes equally well with only cuDNN.

In addition, cuDNN has limited kernel fusion functionality. Kernel fusion is a well-known optimization technique that reduces the GPU global memory accesses between consecutively executed kernels by merging them into a single kernel. cuDNN supports kernel fusion only for a few DL workload patterns (*e.g.*, a sequence of a convolution, a bias addition, and a ReLU activation). However, it is insufficient for handling various DL operation patterns found in emerging

DL workloads.

To overcome these limitations of cuDNN-based DL frameworks, several DL optimization frameworks have been proposed [13, 14, 15, 16, 17]. Unlike conventional DL frameworks, DL optimization frameworks scan the given DL workload first and apply several off-line optimizations, including kernel fusion techniques. Thanks to these optimizations, some of the DL compilers (*e.g.*, TVM[14], TensorFlow XLA[13], TensorRT[15]) achieve competitive or better performance than cuDNN for certain cases (*e.g.,* small-batched CNN inferences). However, all of them fail to achieve consistent speedup over cuDNN for versatile DL workloads.

The main reason for their relatively low performance compared to cuDNN is the information used for GPU kernel optimizations. To optimize the GPU kernel code, the programmer or the code generator must find the best-performing set of implementation parameters such as the size of a thread block and the tile size of the input feature map. However, generating the code and measuring its performance for all possible sets of parameters is almost impossible because the parameter search space is too large to handle.

Meanwhile, widely used DL frameworks also fail to adapt to increased model scale. There are two reasons for this. The first is the GPU-memory-centric design of the DL frameworks. When using a GPU for training, DL frameworks allocate all tensors in GPU memory by default. However, because the sizes of the DL models have increased, it has become impossible to use this approach owning to an out-of-memory (OOM) error.

The other reason is the limited support for the parallelization of the training workload. Widely used DL frameworks support data parallelism only when using multiple GPUs. In this case, it is impossible to train a huge DNN model because all the parameters should be duplicated in the memory of each GPU.

To overcome these limitations, several large-scale model-targeting frameworks have been proposed[18, 19, 20, 21, 22, 23]. These frameworks support the training of large-scale models by offloading the tensors to CPU memory during the training process (*e.g.*, Capuchin[19], ZeRO-offload[22], TFLMS[18]), or by supporting a versatile type of parallelism, such as weight partitioning (*i.e.*, intra-layer model parallelism) and pipelining (*i.e.*, inter-layer model parallelism).

However, these approaches still show a limited functionality in terms of the supported model type[21, 20, 22] or supported hardware systems[23, 22, 19, 18]. The reason for this limitation is the extremely large search space of the design choices. Similar to the cases of the kernel code optimization, a number of parameters and design choices exist for implementation of the model training. The previous approaches manually select the parameters for a specific model or a particular hardware system. However, to obtain a good performance, a generalized method that finds the best-performing parameters for the given target model and the target architecture should be devised.

To this end, we propose performance-model-driven approaches to handle the DL workload versatility and to handle the out-of-GPU-memory scale model training. To manage the versatility, we propose DeepCuts, an optimized kernel generator for versatile GPU workloads. The approach of DeepCuts has much in common with those of expert programmers. DeepCuts has two significant differences when compared to other existing DL optimization frameworks. As one difference, DeepCuts directly uses the information of the underlying architecture (*e.g.*, the size of GPU shared memory) to build a performance estimator. As the other, instead of estimating the actual performance, DeepCuts estimates the upper bound of the performance and uses it to prune definitely slow cases because estimating the actual performance is extremely difficult and inaccurate

for many cases.

For the set of DL operations in a given DL model, DeepCuts searches for the best-performing implementation parameters of GPU kernels for the DL operations. During the search, it considers the kernel fusion and prunes definitely slow cases using the architecture-aware performance estimation model. Based on the selected sets of implementation parameters, it generates GPU kernels using data-flow graphs. It then executes the generated kernels and selects the best performing kernel for each DL operation or fused DL operation in the DL workload.

To handle the out-of-GPU-memory scale model training, we propose BigCuts, a large-scale-model-targeting DL framework for storage-enabled training on a GPU cluster. BigCuts searches for the best workload scheduling and the parallelization for the given system architecture and target DL workload. Similar to DeepCuts, BigCuts uses a performance estimation model to find the best-performing set of parameters and design choices.

The major contributions of this thesis are as follows:

- DeepCuts provides a competitive or better performance than cuDNN on versatile DL workloads, including small-batch inference, large-batch inference, and training. To the best of our knowledge, DeepCuts is the first framework that achieves both a high performance and versatility for DL workloads.

- We propose a novel performance-model-driven code generation algorithm that considers two critical factors for a performance improvement, fusion, and parameter search, simultaneously. The algorithm searches for the best-performing kernel fusion method by considering the best-performing implementation parameters within a relatively small amount of time ow-

ing to the simple but powerful performance model.

- Unlike cuDNN that is a closed-source library, DeepCuts reveal how the state-of-the-art performance can be achieved using only simple and easy-to-understand techniques without relying on any human efforts.

- We evaluate DeepCuts using widely used DNN models, including CNNs, RNNs, and transformer-based models on an NVIDIA V100 and RTX 2080 GPUs. Over cuDNN, DeepCuts achieves speedups up to 1.70 and 1.89 for small-batch inference, 2.06 and 2.52 for large-batch inference, and 1.09 and 1.10 for training on V100 and RTX 2080, respectively.

- We also compare DeepCuts with three existing DL optimization frameworks for GPUs, TVM, TensorFlow XLA, and TensorRT. It achieves comparable performance to TVM with the code generation time reduced by 13.53×. It achieves speedups of 1.25, 1.48, and 1.36 over TVM, TensorFlow XLA, and TensorRT, respectively, when including cuDNN primitives in the kernel selection process.

- BigCuts is the first framework that support out-of-GPU-memory scale model training on a GPU cluster built using COTS components. Unlike previous state-of-the-art approaches optimized for a specific cluster architecture (*e.g.*, DGX SuperPOD), BigCuts achieves a good performance on various hardware system configurations.

Chapter 2

Related Work

2.1 Optimization Framework for Single GPU

Widely used conventional DL frameworks, such as PyTorch [5], TensorFlow[6], and MXNet[24] basically map DL operations to cuDNN/cuBLAS primitives or pre-implemented CUDA kernels. Although this approach achieves good performance for traditional workloads, such as large-batch CNN inference, it is not suitable for handling versatile workloads and for kernel fusion.

There are four major DL optimization frameworks that generates or uses workload-specific GPU kernels: TensorFlow XLA[13] from Google, TensorRT[15] from NVIDIA, TVM[14], and Tensor Comprehensions[17]. Theses framework can be categorized into two categories: frameworks that heavily rely on hand-tuned kernels, and frameworks that generate kernels using ML-based optimizations.

While the first category's frameworks show good performance for some specific models, they are not applicable for versatile models. TensorFlow XLA[13]

and TensorRT[15] are in this case. TensorFlow XLA scans DL operations and generates a fused kernel for a sequence of simple DL operations (*e.g.*, elementwise-operations). However, for time-consuming complex operations(*e.g.*, convolution), it just rely on cuDNN primitives. TensorRT[15] uses cuDNN-like pre-implemented GPU kernels for some specific patterns of DL operations (*e.g.*, a convolution operation followed by a batch normalization operation and a ReLU operation). Although TensorRT shows good performance for the models that use the supported patterns, it fails to achieve good performance for the models that do not have the patterns.

On the other hand, frameworks in the second category propose flexible code generation techniques (*i.e.*, applicable to versatile models), while their performance is relatively lower than those of the hand-tuned libraries. TVM[14] and Tensor Comprehensions are in this case. TVM scans a given DL workload, fuses its DL operations using predefined rules, and generates optimized GPU kernels using a machine-learning-based performance model and simulated annealing. TVM achieves state-of-the-art performance for small-batched CNN inferences. However, it fails to achieve cuDNN-competitive performance for large-batched CNN inferences. Tensor Comprehensions[17] is a domain-specific language that specifies DL operations. It automatically tunes the kernels using a genetic algorithm. Its performance for commonly-used DL operations (*e.g.*, convolution) is worse than those of cuDNN, TVM, and DeepCuts.

DeepCuts achieves the strengths of both categories; flexibility and performance. In terms of flexibility, DeepCuts supports versatile types of DL operations because it uses techniques that are not limited to specific DL operations. In terms of performance, DeepCuts achieves cuDNN-competitive or better performance thanks to its novel optimization techniques that rely on an architecture-aware performance estimation model.

Table 2.1: Comparison of DL Optimization Frameworks

		DeepCuts	TVM (autoTVM)	TensorFlow XLA	TensorRT
Models supported	CNNs	✔	✔	✔	✔
	RNNs	✔		✔	✔
	TMLPs[a]	✔		✔	✔
Workload types	Inference	✔	✔	✔	✔
	Training	✔		✔	
Convolution algorithms	Winograd	✔	✔	✔	✔
	FFT	✔		✔	✔
SOTA[b] performance	CNNs	✔	partial	partial	partial
	RNNs	✔		✔	
	TMLPs	✔		✔	✔
Key idea for GPU kernel optimization		Architecture-aware performance model	ML-based performance model	Manually tuned library	Manually tuned library

[a] TMLP: transformer-based MLP. [b] SOTA: state-of-the-art.

Table 2.1 compares the existing DL optimization frameworks with DeepCuts. Tensor Comprehensions is not included in this table because full-model implementations using Tensor Comprehensions have not been provided yet. DeepCuts supports various kinds of models and algorithms. It achieves state-of-the-art performance for all of them. TVM does not achieve good performance for large-batch inference, and TensorFlow XLA and TensorRT are not suitable for small-batch inference.

There are a few other frameworks that we do not directly compare the performance. Intel nGraph[16] supports various features, including optimizations for training, code generation for complex DL operations, and fusion. However, nGraph mainly targets CPUs. We observe that GPU support of nGraph is not working in the current release (version 0.29.0) and we cannot find any evaluation

results available.

TASO[25] focuses on transforming the input computation graph of tensor operations to obtain better performance (*e.g.*, substituting two small matrix multiplications with a large, faster single matrix multiplication). TASO and DeepCuts have some similarities in a few aspects (*e.g.*, flexible patterns of fusion). However, we do not directly compare their performance because there is a significant difference between the two frameworks. There are two reasons for this. One is that the level of optimizations is quite different between DeepCuts and TASO. TASO does not focus on generating GPU kernels. Instead, it focuses on rewriting the computation graph to better exploit a given primitive library (*e.g.*, cuDNN primitives). On the other hand, DeepCuts mainly focuses on generating optimized GPU kernels for the given computation graph. The other reason is that DeepCuts does not change the amount of total computation while TASO does. DeepCuts aims to perform architecture-aware optimization without changing the total amount of computation itself (*i.e.*, FLOPS). TASO performs various high-level graph-substitutions that alter the total amount of computation for many cases. Both approaches are beneficial and can be cooperatively used but are not appropriate for direct comparison.

Apart from the DL optimization frameworks, some previous studies[26, 27, 28, 29, 30] propose techniques to find optimized code for some important DL operations (*e.g.*, matrix multiplication and convolution). Widely used BLAS libraries[26, 27] rely on predefined kernels with the optimal set of implementation parameters (*e.g.*, tile size). These parameters are found by trying every possible case in a brute-force manner. OpenTuner[29] and KernelTuner[28] use evolutionary algorithms to find the optimal parameters. Pfaffe *et al.*[30] combines evolutionary algorithms and polyhedral models to generate the optimal kernel code.

Previous kernel fusion studies inspired and affected DeepCuts. Alwani *et al.*[31] propose a fusion technique that keeps the intermediate result of the previous kernel in on-chip memory and uses it for the next kernel. Jung *et al.*[32] split a batch normalization layer into two phases and fuse them into the previous and next convolution layers of the batch normalization layer to accelerate CNN training. The kernel fusion technique used by DeepCuts is significantly different from the previous approaches in that DeepCuts searches for the best-performing implementation and architecture parameters for fusion candidate kernels. Then it relies on a performance estimation model to determine their actual fusion.

2.2 Optimization Framework for GPU Cluster

PyTorch [5] and TensorFlow[6] support modules for distributed training on a GPU cluster [33, 34]. Users can easily parallelize their training scripts with a few lines of modifications, using the provided module. In addition, Horovod [**?**], a third-party library of PyTorch and TensorFlow, enables users to parallelize their PyTorch/TensorFlow scripts without any modification. However, these frameworks and modules support data parallelism only. Since we need to duplicate all the parameters to each GPU when we use data parallelism, it is impossible to train a OOGM model using data parallelism only. To exploit other types of parallelism (*e.g.*, weight partitioning, pipelining) using PyTorch or TensorFlow, the programmer needs to implement parallel training scripts manually, which is much harder than using provided modules.

Mesh-TensorFlow [20] provides an interface that enables users to parallelize their training workload in an arbitrary way on a GPU cluster. Users can select a dimension of a tensor to be divided (*e.g.*, batch-dimension, the height of the input image, etc.), and the framework automatically divides the data and the

computation to multiple GPU. However, to obtain a good performance, the programmer needs to fully understand the model structure and the overheads of parallelization (*e.g.*, communication overhead).

Megatron and DeepSpeed [21] parallelizes OOGM transformer model training using multiple types of parallelism collaboratively (*i.e.*, weight partitioning and pipelining). These frameworks provide transparent programming interfaces that enable users to train the model without concerning about the details of the parallelization. However, there are two problems for these frameworks. The first problem is the required number of GPUs. Since these frameworks store every tensor in GPU memory, these require a huge number of GPUs to train a huge OOGM model (*e.g.*, more than 512 GPUs to train transformer model with 1 trillion parameters). The other problem is the target hardware system. These frameworks require extremely high GPU-to-GPU communication bandwidth, which can be obtained only for the DGX-2 and DGX Superpod systems[35] that use NVLink and NVSwitch in them. In this reason, the training throughput is extremely degraded for other types of hardware systems that use COTS GPUs and PCIe only.

ZeRO-infinity[23] solved the first problem using the storage system. It stores tensors in the NVMe, and transfers them to GPU memory when they are needed. However still, the second problem remains. Since ZeRO-infinity requires a system with extremely high bandwidth of GPU-to-GPU communication, it is hard to obtain good performance on a non-DGX systems.

Chapter 3

Optimization Framework for a Single GPU

3.1 Overall Structure of DeepCuts

Figure 3.1 shows the overall workflow of DeepCuts. It takes a computational graph of tensor operations and generates a set of GPU kernels. The input graph is an acyclic graph that describes the computation and data flow of a DNN model. An edge of the graph represents a tensor of data and a node represents a tensor operation (*e.g.*, convolution). When the graph is implemented for a GPU, each node corresponds to a GPU kernel call or a DNN library function call (*e.g.*, cuDNN[12]). The input graph is similar to the computational graph of PyTorch[5] or TensorFlow[6].

DeepCuts uses the input graph generated from the PyTorch script of a DNN model. We modify the PyTorch source code to emit the information of tensor operations executed. We use this information to reconstruct the input computation graph of DeepCuts.

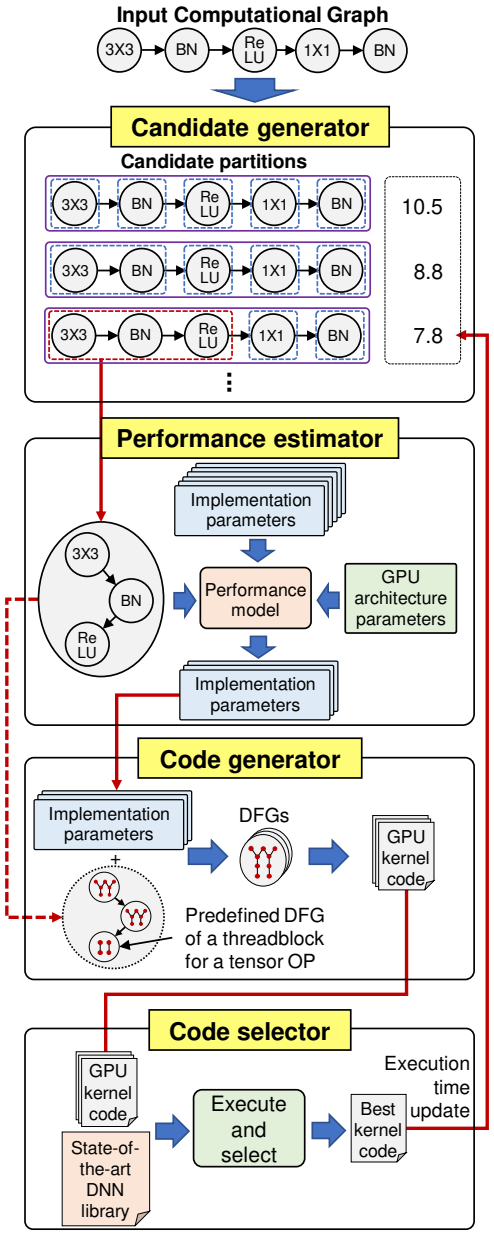


Figure 3.1: Overall workflow of DeepCuts. DeepCuts takes the whole computational graph of given workload and generates corresponding GPU kernels.

```
Input: Initial partition Q, each subset is a singleton set
Output: Set of partitions R
R ← ∅;
Evaluate(Q);
R ← R ∪ {Q};
while there exist an unmarked partition P in R do
    mark P;
    foreach consecutive subsets X and Y in P do
        Q ← a partition generated from P by merging X and Y;
        Evaluate(Q);
        if Q performs better than P then
            R ← R ∪ {Q};
        end
    end
end
```

Algorithm 1: Partition generation and selection.

As shown in Figure 3.1, DeepCuts consists of four modules: *candidate generator*, *performance estimator*, *code generator*, and *code selector*.

The candidate generator partitions the set of nodes (*i.e.*, tensor operations) for a given DL workload. A partition of the nodes is a grouping of the nodes into non-empty subsets in such a way that every node is included in exactly one subset. Each subset contains a single node or multiple consecutive (*i.e.*, connected by edges) nodes.

DeepCuts generates a single GPU kernel for each subset. Putting multiple nodes into a single subset implies that the corresponding GPU kernels to the nodes are fused into a single kernel for the entire subset.

The candidate generator generates multiple partitions as code generation candidates, evaluates each of them, and identifies a partition with the best performance. To check the performance of each subset in a partition, the candidate

generator uses other modules: performance estimator, code generator, and code selector.

For a given partition P, the task performed by the performance estimator, code generator, and code selector can be abstracted as a procedure $Evaluate(P)$. $Evaluate(P)$ returns the execution time of the given partition P. Its function is defined as follows:

- For each subset S of P, the performance estimator searches for kernel implementation parameters, such as the number of total kernel threads, the thread block size, and the number of output features per thread. It uses a performance estimation model and searches for parameter combinations that make S to be in the top $T\%$ (*e.g.*, $T = 1$ and T is determined empirically) of the best performing combinations of implementation parameters. The performance estimator assumes fusing the nodes in S if S is not a singleton set.

- For each group of kernel implementation parameters found for S, the code generator generates a GPU kernel.

- The code selector executes the generated kernels with a random input and measures their execution time. It selects a kernel with the best execution time for S. If the corresponding state-of-the-art DNN library function, such as cuDNN[12], is better than the selected kernel, the library function is selected as the best performing kernel. It also updates the total execution time of P with the best kernel execution time.

In principle, to find an optimal partition, the candidate generator needs to consider all possible partitions of the given nodes. However, the number of possible partitions grows exponentially as the number of nodes increases. For

Table 3.1: Types of Tensor Operations Handled by DeepCuts

Category	Node type
Simple	Unary element-wise operations (*e.g.*, ReLU, tanh, sigmoid), Binary element-wise operations (*e.g.*, matrix addition, matrix subtraction), Winograd transformation, 2D FFT, Depthwise separable convolution, Pooling, Batch normalization, Layer normalization
Complex	Direct convolution, Matrix multiplication

instance, the computational graph of the inference workload of VGG-16[36] has 53 nodes. In this case, the number of possible partitions is more than 2^{52}.

Partitioning algorithm. Instead of considering every possible partition, the candidate generator incrementally generates more partitions from the current partition if the current partition has potential speedup over existing partitions. The sketch of the partition generation and selection algorithm is shown in Algorithm 1.

The algorithm is continued until no new partition is generated. The candidate generator memorizes the generated code and the execution time of each subset in the partitions in R. In the end, the candidate generator compares the execution times of partitions in R and selects the best performing partition and its code.

For instance, assuming the given input computation graph is $A \rightarrow B \rightarrow C$, where A, B, and C are tensor operations. Then, the initial partition Q in Algorithm 1 would be $\{\{A\},\{B\},\{C\}\}$, and the initial R would be $\{\{\{A\},\{B\},\{C\}\}\}$. For simplicity, we assume that every fusion gives speedup in this example. Then, after the first iteration of while loop, R becomes $\{\{\{A\},\{B\},\{C\}\},\{\{A,B\},\{C\}\},\{\{A\},\{B,C\}\}\}$. After the second iteration, R becomes $\{\{\{A\},\{B\},\{C\}\},\{\{A,B\},\{C\}\},\{\{A\},\{B,C\}\},\{\{A,B,C\}\}\}$. Af-

ter the next iteration, the iteration stops because no more partition is generated (*i.e.*, all partitions are marked). Then the algorithm returns the set of partitions R.

Note that, although the proposed algorithm does not enumerate all of the possible candidates, it enumerates almost every candidate that improves the performance. For instance, given three consecutive subsets X, Y, and Z, the algorithm try to fuse all of them (*i.e.*, put all of them into one subset) only when fusing two of them ($X + Y$ or $Y + Z$) is found to be beneficial. However, we empirically observe that $X + Y + Z$ very rarely improves performance when both $X + Y$ and $Y + Z$ degrade the performance. Thus, the algorithm excludes this case.

We explain the performance estimation model in detail in Section 3.2. The detailed code generation algorithm is explained in Section 3.3 and Section 3.4.

Target tensor operations. Table 3.1 summarizes the types of nodes (*i.e.*, tensor operations) that DeepCuts handles. They are categorized into two classes: *complex* and *simple*. The operations whose computational complexities are proportional to the input size are simple operations. They are often memory intensive. The remaining operations are complex operations. They are time-consuming and compute-intensive operations, such as direct convolution and matrix multiplication.

3.2 Performance Estimation Model

The goal of using the performance estimation model in DeepCuts is *not* to estimate the exact performance. Instead, DeepCuts aims at pruning the cases that definitely slow down using the performance estimation model. The model extends the roofline model[37] to estimate the *upper bound* of the performance

of a tensor operation for given GPU architecture parameters and kernel implementation parameters.

DeepCuts assumes that the data in the DNN model are represented in the single-precision floating-point format and stored in 4D tensors in the $NCHW$ format in the GPU global memory.

Table 3.2: Kernel Implementation Parameters Obtained from Operation Definition

Symbol	Parameters
N, C, K, H, W	Batch size(N), numbers of input/output channels(C, K), height/width of the output feature map(H, W)
F_W, F_H, P_W, P_H, S_W, S_H	width-wise and height-wise sizes of the filter (F), padding (P), and stride (S)

Table 3.3: Kernel Implementation Parameters to Search for

Symbol	Parameters
N_{block}, K_{block}, H_{block}, W_{block}	Number of output images in the batch (N_{block}), number of output channels (K_{block}), width (W_{block}) and height (H_{block}) of the output feature map that a thread block computes
N_{thread}, K_{thread}, H_{thread}, W_{thread}	Number of output images in the batch (N_{thread}), number of output channels (K_{thread}), width (W_{thread}) and height (H_{thread}) of the output feature map that a thread computes
C_{input}	Number of input channels processed by a thread block per iteration
$FORMAT_{shared}$	Input data layout in shared memory

3.2.1 Kernel Implementation Parameters

The kernel implementation parameters considered in DeepCuts are described in Table 3.2 and Table 3.3. The parameters listed in Table 3.2 (N, C, K, H, W, F_W, F_H, P_W, P_H, S_W, and S_H) describe a convolution operation. The same parameters can be used to describe other operations. For example, matrix

multiplication is considered as a convolution operation whose H and W are set to one. These parameters are fixed and not changed during the optimization because they are derived from the definition of a tensor operation itself.

The parameters listed in Table 3.3 are variable parameters. DeepCuts tries to find a well-performing combination of variable parameters during the optimization. The first and second rows in Table 3.3 describe how the computation is distributed across multiple threads and thread blocks.

The performance estimation model assumes that the generated kernel code has two phases: *fetch* and *compute*. In the fetch phase, all threads in a thread block cooperatively load input data from the global memory and store them to the shared memory. In the compute phase, the threads in the thread block perform computation accessing the data stored in the shared memory.

For some operations (*e.g.*, convolution), it is impossible to fetch all input data to a thread block on shared memory at once. In this case, DeepCuts generates code that iterates over the fetch phase and the compute phase. We call this a *looped operation*. In this case, the input data are split along the channel dimension. C_{input} channels of the input data are fetched to the shared memory in each iteration.

$FORMAT_{shared}$ is a parameter that describes the data layout in the shared memory. DeepCuts considers 24 different combinations of N, C, H, and W: $NCHW$, $NCWH$, ..., $WHCN$.

DeepCuts first makes the implementation parameter search space smaller using the following rules:

- N_{thread}, K_{thread}, H_{thread}, and W_{thread} are set to powers of two.
- N_{block}, K_{block}, H_{block}, and W_{block} are multiples of N_{thread}, K_{thread}, H_{thread}, and W_{thread}, respectively.

3.2.2 Performance Limiting Factors

Based on our experiences of manual GPU kernel implementations for tensor operations with various implementation parameters, we observe that there are four major architecture-related performance limiting factors: (1) *global memory bandwidth*, (2) *shared memory latency*, (3) *workload imbalance between streaming multiprocessors (SMs)*, and (4) *limitation of hardware resources*.

We build a performance model based on this observation. From the kernel implementation parameters, the model checks if each performance limiting factor limits the kernel's performance. To do this, the model computes a metric value for each limiting factor ($GMRatio$, $SMRatio$, $WBRatio$, and $COEF_r$, explained in detail in the following paragraphs). Then, it computes the performance upper bound of the kernel using these metric values. Based on this information, DeepCuts can prune definitely-slow kernels without actually generating and executing them.

Global memory bandwidth. An *operational intensity* is the amount of compute operations per byte of global memory traffic[37]. For a given tensor operation, the operational intensity of its GPU kernel varies a lot depending on the implementation parameters.

Table 3.4: GPU Architecture Parameters

Symbol	Parameters
NUM_{SM}	Number of streaming multiprocessors.
$PERF_{peak}$	Peak performance of single-precision floating-point operations
BW_{global}	Bandwidth of the global memory
$TRANS_{global}$	Maximum number of input elements in a global memory transaction
LAT_{shared}	Latency of the shared memory
MAX_{shared}	Maximum size of shared memory allocation for a thread block
MAX_{thread}	Maximum number of threads in a thread block

Table 3.5: Amount of Computation per Thread Block

Algorithm	Formula
Direct convolution	$2 \cdot N_{block} \cdot C \cdot K_{block} \cdot H_{block} \cdot W_{block} \cdot F_H \cdot F_W$
Matrix multiplication	$2 \cdot N_{block} \cdot C_{block} \cdot K_{block}$
Batch normalization	$3 \cdot N_{block} \cdot K_{block} \cdot H_{block} \cdot W_{block}$
Winograd transform	$2.25 \cdot N_{block} \cdot K_{block} \cdot H_{block} \cdot W_{block}$
Pooling	$N_{block} \cdot K_{block} \cdot H_{block} \cdot W_{block} \cdot F_H \cdot F_W$
Element-wise operation	$N_{block} \cdot K_{block} \cdot H_{block} \cdot W_{block}$

Table 3.6: Shared Memory Load Operations per Thread

Algorithm	Formula
Direct convolution	$N_{thread} \cdot C \cdot (\lfloor (H_{thread} + F_H - 2 \cdot P_H)/S_H \rfloor + 1) \cdot (\lfloor (W_{thread} + F_W - 2 \cdot P_W)/S_W \rfloor + 1) + (K_{thread} \cdot C \cdot F_W \cdot F_H)$
Matrix multiplication	$N_{thread} \cdot C_{thread} + C_{thread} \cdot K_{thread}$
Batch normalization	$N_{thread} \cdot K_{thread} \cdot H_{thread} \cdot W_{thread} + 3 \cdot K_{thread}$
Winograd transform	$N_{thread} \cdot K_{thread} \cdot H_{thread} \cdot W_{thread}$
Pooling	$N_{thread} \cdot K_{thread} \cdot H_{thread} \cdot W_{thread} \cdot F_H \cdot F_W$
Element-wise unary op.	$N_{thread} \cdot K_{thread} \cdot H_{thread} \cdot W_{thread}$
Element-wise binary op.	$2 \cdot N_{thread} \cdot K_{thread} \cdot H_{thread} \cdot W_{thread}$

To compute the operational intensity of a thread block, DeepCuts obtains the amount of computation in a thread block using the formulae listed in Table 3.5. It also computes the number of global memory transactions in a thread block using the formulae listed in Table 3.7. The formulae listed in Table 3.5 and Table 3.7 uses kernel implementation parameters listed in Table 3.3 and the target GPU architecture parameters listed Table 3.4. Then, it computes the operational intensity of a thread block, denoted OI_{TB}:

$$OI_{TB} = COMP_{block}/(SIZE_{element} \cdot TRANS_{global} \cdot NUM_{trans})$$

where $COMP_{block}$ is the amount of computation in the thread block, $SIZE_{element}$ is the size of an input element in bytes (4 for a single-precision

Table 3.7: Global Memory Transactions per Thread Block

Algorithm	Formula
Direct convolution	$N_{block} \cdot C \cdot (\lfloor (H_{block} + F_H - 2 \cdot P_H)/S_H \rfloor + 1) \cdot (\lceil (\lfloor (W_{block} + F_W - 2 \cdot P_W)/S_W \rfloor + 1)/TRANS_{global} \rceil) + \lceil (K_{block} \cdot C \cdot F_W \cdot F_H)/TRANS_{global} \rceil$
Matrix multiplication	$(N_{block} \cdot \lceil C_{block}/TRANS_{global} \rceil + C_{block} \cdot \lceil K_{block}/TRANS_{global} \rceil$
Batch normalization	$N_{block} \cdot K_{block} \cdot H_{block} \cdot (\lceil W_{block}/TRANS_{global} \rceil) + 3 \cdot \lceil (K_{block}/TRANS_{global}) \rceil$
Winograd transform	$N_{block} \cdot K_{block} \cdot H_{block} \cdot (\lceil W_{block}/TRANS_{global} \rceil)$
Pooling	$N_{block} \cdot K_{block} \cdot (\lfloor (H_{block} + F_H - 2 \cdot P_H)/S_H \rfloor + 1) \cdot \lceil (\lfloor (W_{block} + F_W - 2 \cdot P_W)/S_W \rfloor + 1)/TRANS_{global} \rceil$
Element-wise unary op.	$N_{block} \cdot K_{block} \cdot H_{block} \cdot \lceil W_{block}/TRANS_{global} \rceil$
Element-wise binary op.	$2 \cdot N_{block} \cdot K_{block} \cdot H_{block} \cdot \lceil W_{block}/TRANS_{global} il$

floating-point value), and NUM_{trans} is the number of global memory transactions to load the input data for the thread block.

Since DeepCuts assumes that every thread block performs the same amount of computation, the operational intensity of a kernel is the same as that of a thread block.

To estimate the effect of the global memory accesses, we define a simple metric $GMRatio$ as follows:

$$GMRatio = Min(1, OI_{TB}/(R_{peak}/BW_{global})$$

where R_{peak} is the theoretical peak performance of the GPU in single-precision floating-point operations per second and BW_{global} is the global memory bandwidth of the GPU in bytes per second. R_{peak}/BW_{global} is the theoretical peak operational intensity. The value of $GMRatio$ varies between 0 and 1. If $GMRatio = 1$, the global memory bandwidth does not bound the performance.

For example, Figure 3.2 shows the roofline model[37] of NVIDIA Tesla V100. It indicates that, when the operational intensity is less than 15.6, the

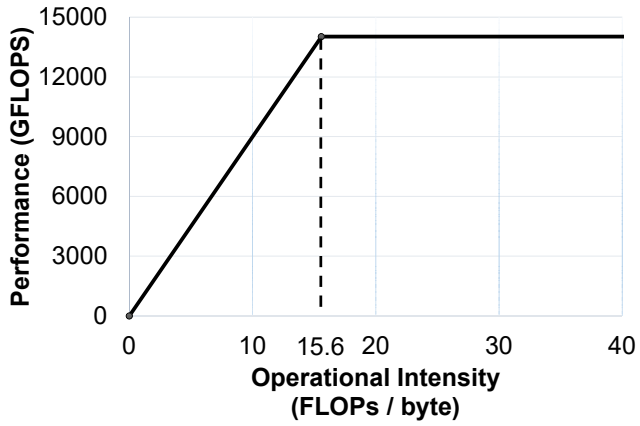


Figure 3.2: Roofline model of Nvidia Tesla V100.

performance is bounded by the global memory bandwidth resulting in a lower $GMRatio$ than 1.

Shared memory latency. Given a set of architecture and implementation parameters, it is very difficult to estimate the effect of the shared memory latency on the performance. Instead, similar to the case of global memory accesses, we use a simple model to estimate the effect of the shared memory latency on a kernel. DeepCuts computes the ratio of the number of pure computation operations to the number of shared memory load operations in a thread. Then it compares the result with the shared memory latency.

To do so, we define $SMRatio$ as follows:

$$SMRatio = Min(1, (COMP_{thread}/N_{load})/(LAT_{shared} \cdot COEF_{bc}))$$

where $COMP_{thread}$ is the number of pure compute operations, N_{load} is the number of shared memory load operations, L_{shared} is the latency of a shared memory load operation in cycles, and $COEF_{bc}$ is the coefficient that shows the degree of shared memory bank conflicts. $SMRatio$ measures how much of the

shared memory latency is hidden by other compute operations. We assume that the latency of a compute operation is one cycle.

To obtain $COEF_{bc}$, we count the number of possible bank conflicts. Since the access pattern for the shared memory is determined statically, we can count the number of conflicts before generating the kernel. For each shared memory load operation, DeepCuts checks the addresses accessed by a warp, checks if bank conflicts occur, and calculates the expected number of cycles for each shared-memory load in the same way as NVIDIA visual profiler[38]. If the expected number of bank conflicts is 0, we set $COEF_{bc}$ as 1. For example, if a thread performs 10 floating-point operations per shared memory load, where the shared memory load latency is 20 cycles and there are no bank conflicts, then 10 cycles out of 20 cycles are hidden. In this case, $SMRatio = 0.5$. If the number of operations is bigger than 20, $SMRatio$ becomes 1. In this case, the shared memory latency is not a performance limiting factor. To compute $SMRatio$, we need to compute $COMP_{thread}$ and N_{load} for each different type of operations. $COMP_{thread}$ is computed using the formulae listed in Table 3.5, by replacing N_{block}, K_{block}, H_{block}, W_{block} with N_{thread}, K_{thread}, H_{thread}, W_{thread}. Formulae for N_{load} is listed in Table 3.6

Workload imbalance across SMs. Since DeepCuts assumes that each thread block performs the same amount of computation, there is no workload imbalance across thread blocks. However, the workload imbalance does appear when the number of thread blocks is not a multiple of the number of SMs.

We formulate this inefficiency using $WBRatio$. Similar to $GMRatio$ and $SMRatio$, the value of $WBRatio$ also varies between 0 and 1. The higher $WBRatio$, the higher the expected performance. The number of thread blocks

and $WBRatio$ is computed as follows:

$$N_{TB} = (N/N_{block}) \cdot (K/K_{block}) \cdot (H/H_{block}) \cdot (W/W_{block})$$

$$WBRatio = 1 - ((N_{TB}N_{SM})/N_{SM})/\lceil (N_{TB}/N_{SM}) \rceil$$

where N_{TB} is the number of thread blocks and N_{SM} is the number of SMs.

Limitation of hardware resources. When the kernel implementation parameters require more hardware resources than the available resources provided by the GPU, it is impossible to execute the generated kernel code. The performance model prunes these cases using $COEF_r$, which is a binary value set to 1 or 0. $COEF_r$ is set to 1 when the implementation parameters require a feasible amount of hardware resources. Otherwise, it is set to 0.

To compute $COEF_r$, the model uses the number of threads (NUM_{thread}) in a thread block and the amount of shared memory usage ($SIZE_{shared}$). They can be directly computed from the implementation parameters shown in Table 3.3. $COEF_r$ is computed as follows:

$$COEF_r = \begin{cases} 1 & NUM_{thread} < MAX_{thread} \\ & \quad \text{and} \quad SIZE_{shared} < MAX_{shared} \\ 0 & otherwise \end{cases}$$

3.2.3 Estimating the Upper Bound

By multiplying $GMRatio$, $SMRatio$, $WBRatio$, and $COEF_r$ computed for a tensor operation, DeepCuts obtains a metric, denoted PUL, for the upper bound of the performance of the tensor operation:

$$PUL = GMRatio \cdot SMRatio \cdot WBRatio \cdot COEF_r$$

PUL represents the ratio of the expected maximum performance of the tensor operation to the peak theoretical performance of the target GPU. The value of

PUL varies between 0 and 1. For example, suppose that $PUL = 0.7$ for given implementation parameters. In this case, we cannot obtain more than 70% of the theoretical peak performance of the GPU with the given implementation parameters.

DeepCuts computes PUL of every possible combination of implementation parameters. Then it picks the combinations whose PUL is in the top 1% of the best performing combinations. These sets of parameters are fed to the next stage, code generation (Figure 3.1).

Because the search space of the kernel implementation parameters is too large, it is hard to prove that DeepCuts always finds the best parameters for every case. Thus, instead of checking every possible tensor operations, we test commonly-used tensor operations (*e.g.*, 3x3 convolution in ResNet[9], matrix multiplication in BERT[4]) with a sufficient number of implementation parameters (more than 100,000). We do not find any case that the model prunes the best-performing set of parameters.

3.2.4 Shared-Memory-Level and Register-Level Fusion

When a subset of nodes in a partition contains two consecutive nodes, the performance estimation model searches for implementation parameters by considering their fusion. Note that if the subset is not a singleton set, it always contains two consecutive nodes by the partitioning algorithm described in Section 3.1.

There are two ways of fusing two consecutive operations (say the predecessor and the successor): *at the shared memory level* and *at the register level*. The register-level fusion keeps the intermediate result in the registers, while the shared-memory-level fusion stores the intermediate result between the two operations in the shared memory.

DeepCuts uses a simple, deterministic algorithm for selecting the way of fusion. The register-level fusion is applied to the following two cases: One is a pair of two simple operations (*e.g.*, an element-wise operation followed by ReLU). The other is a pair of one simple operation and one complex operation (*e.g.*, a convolution operation followed by bias addition). When multiple threads in the succeeding operator should use the result of a thread's computation in the preceding operator, shared-memory-level fusion is used (*e.g.*, a convolution operation followed by another convolution operation).

For a pair of operations that are fused in the register-level, the performance estimation model computes PUL for a single, fused operation with the following rules:

- NUM_{trans} and N_{load} of the fused operation is the same as those of the predecessor.
- $COMP_{block}$ and $COMP_{thread}$ of the fused operation is the sum of those of the predecessor and the successor.

$COMP_{block}$ is the amount of computation in a thread block and $COMP_{thread}$ is the amount of computation in a thread.

For a pair of operations that are fused in the shared-memory-level, the performance model computes the values of PUL for both operations as usual but with the following three constraints:

- NUM_{trans} for the successor is set to 0 (*i.e.*, $GMRatio$ is set to 1) because all input data to the successor that are generated by the predecessor are stored in the shared memory.
- The predecessor's output size per thread block should match the input size of the successor. The estimation model checks the implementation

parameters, and discard unmatched cases.

- When the successor is a complex operation, its C_{input} should be equal to C. This implies that all input data of the successor is assumed to be stored in the shared memory.

The model determines to fuse two consecutive operations that satisfy the following conditions:

- Register-level fusion of two simple operations makes a simple operation.
- Register-level fusion of a simple operation and a complex operation makes a complex operation.
- Two complex operations can only be fused at the shared memory level.

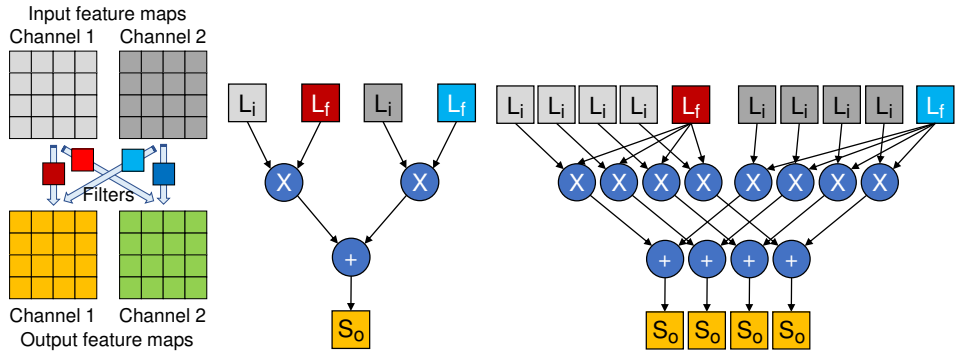


Figure 3.3: Baseline DFG construction. (a) 1×1 convolution operation with 2 input channels and 2 output channels. (b) The DFG of an output pixel when $C_{input} = 2$. (c) The baseline DFG for a thread block when $C_{input} = 2$.

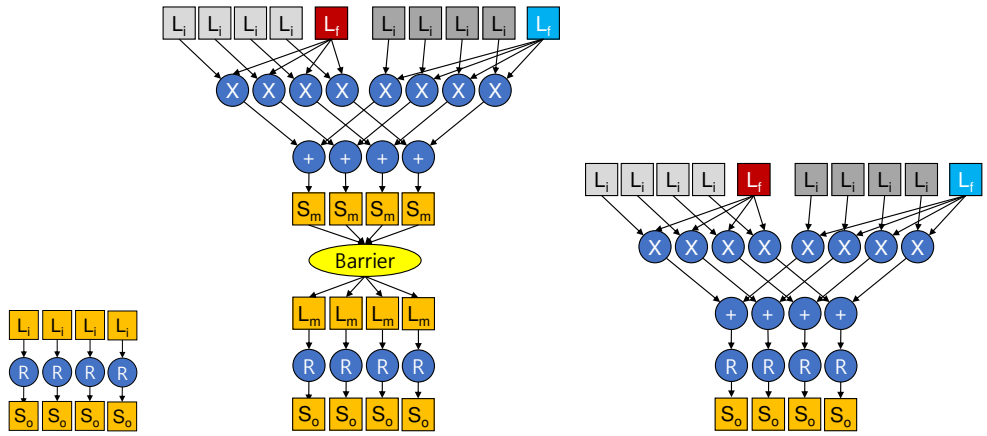


Figure 3.4: DFG concatenation. (a) Baseline DFG for ReLU operation. (b) Shared-memory-level fusion of a 1×1 convolution and a ReLU operation. (c) Register-level fusion of a 1×1 convolution and a ReLU operation.

3.3 Data-Flow Graph Generation

DeepCuts relies on a data-flow-graph-based code generation algorithm that uses DFG as an intermediate representation. The DFG generation consists of three steps: (1) *baseline DFG generation*, (2) *DFG concatenation*, and (3) *subgraph extraction for a GPU thread*.

3.3.1 Baseline DFG Generation

The baseline DFG represents the computation in a thread block. DeepCuts has a set of predefined DFGs, each of which represents the computation of a tensor operation. The predefined DFGs assume that all computation is performed using the data stored in the shared memory or registers.

For example, suppose the set S of implementation parameters selected by the performance estimator is given by, S = { N=1, C=2, K=2, W=4, H=4,

$\mathrm{F}_W = 1, F_H = 1,$

$P_W = 0, P_H = 0, S_W = 1, S_H = 1,$

$N_{block} = 1, K_{block} = 1, W_{block} = 2, H_{block} = 2, C_{input} = 2,$

$N_{thread} = 1, K_{thread} = 1, W_{thread} = 2, H_{thread} = 1\}$. The parameters in S indicate that the tensor operation performs a 1×1 convolution operation with 2 input channels and 2 output channels, and the feature map size is 4×4 (Figure 3.3(a)). Then the computation for an output pixel of this tensor operation is represented by a DFG shown in Figure 3.3(b). The nodes with labels L_i, L_f, S_o, X, and $+$ represent a shared memory load operation of an input feature map pixel, a shared memory load operation of a filter element, a shared memory store operation of an output feature map pixel, a binary multiplication, and a binary addition, respectively. This DFG is predefined in DeepCuts for 1×1 convolution operations.

The implementation parameter makes the thread block compute a 2×2 output feature map in a single output channel ($K_{block} = 1$). Thus, DeepCuts copies the DFG of an output pixel in Figure 3.3(b) 4 times and merges the shared memory load operation nodes that access the same memory location and have no predecessor. As a result, we obtain the DFG for a thread block in Figure 3.3(c).

In the case where C_{input} is less than C, the code generator will generate a loop that alternates the fetch and compute phases as mentioned in Section 3.2 for the tensor operation. We call it *a looped complex operation*. In this case, DeepCuts generate two different DFGs. One DFG represents the computation in the loop body (*i.e.*, one iteration of the looped complex operation). The other DFG represents the operation performed at the loop exit (*i.e.*, storing results to the shared memory).

3.3.2 DFG Concatenation for Fusion

When generating the code for a fused operation, DeepCuts generates the baseline DFG for each tensor operation and concatenate them. The resulting DFG is the baseline DFG for the fused operation. Suppose that we generate the code for a 1×1 convolution followed by a ReLU operation. The DFG for the ReLU operation for a thread block is generated as shown in Figure 3.4(a). A node with label R represents a ReLU operation.

There are two ways of concatenation depending on the fusion type (mentioned in Section 3.2.4, at the shared memory level and at the register level) determined by the performance model. Figure 3.4(b) shows the result of of concatenating the two DFGs in Figure 3.3(c) and Figure 3.4(a) for the shared memory-level fusion. Nodes with labels S_m and L_m represent shared memory store and load operations for the intermediate results. The two DFGs are simply concatenated with a barrier operation between them to guarantee memory consistency.

Figure 3.4(c) shows the result of the register-level fusion. Since the intermediate results are kept in registers, nodes with labels S_o and L_i are removed. In the case of looped complex operations, we make a fused DFG using the DFG of the loop exit.

3.3.3 Extracting a Subgraph for a Thread

Since CUDA kernel code describes the computation of a GPU thread, we need to extract a subgraph from the baseline DFG. After building the baseline DFG for a thread block, DeepCuts first partitions the last shared memory store operations (denoted S_o in Figure 3.4) into multiple equal-sized chunks whose size is defined by the implementation parameters (N_{thread}, C_{thread}, H_{thread}, W_{thread}).

Each chunk is assigned to a different thread. DeepCuts follows the edges in the reverse direction from the assigned store operations. It marks all nodes visited in the traversal. After reaching the nodes with no predecessors, DeepCuts removes all unmarked nodes to extract the sub-DFG for a thread.

3.4 GPU Kernel Code Generation

3.4.1 DFG-Based Code Generation

The code generator generates the kernel code based on the sub-DFG for a thread. Note that the GPU kernel code describes the computation done by a thread in a parameterized way.

The code generator first emits code for the header of the kernel function and thread/block ID fetching (*i.e.*, reading *blockIdx* and *threadIdx*). Then, it generates code that fetches data from the global memory to the shared memory. The amount of the input data to be loaded is determined by the implementation parameters selected by the performance estimator. In the generated code, all threads in a thread block cooperatively load the input data in a coalesced manner[39]. The data layout in the shared memory is determined by the implementation parameter $SHARED_{format}$ selected by the performance estimator.

After generating the fetch code, the code generator generates code for computation. It traverses the sub-DFG of a thread in reverse post order (*i.e.*, in the order of the reverse of the list created by post-order traversal) and emits code for each node. The code for each node is straightforward because each of the compute DFG nodes represents a scalar operation.

For a looped complex operation (described in Section 3.3), the code for the looping structure is emitted by the code generator. Then in the loop body, the code for the fetch phase and the code for the compute phase are generated.

DeepCuts generates two variants of code for the looping structure: *normal* and *prefetching*. The *normal* variant execute fetch phase and compute phase as usual. The *prefetching* variant hides the global memory load latency of fetch phase of the next iteration using the compute phase of the current iteration. DeepCuts compares their performance, and selects faster code.

3.4.2 Shared Memory Optimizations

Shared memory optimizations are critical to achieving state-of-the-art performance on GPUs. To find the well-performing optimization technique, we manually implement and test versatile shared memory optimization techniques. We observe that the following methods are effective and make DeepCuts code generator apply these techniques.

Memory-based index function. Since a CUDA kernel is written in SPMD style (*i.e.*, in a data-parallel manner), to make each thread correctly access its data in a data-parallel manner, an index function is required. It takes thread ID as input and returns a corresponding index to the data element the thread accesses.

DeepCuts implements the index function in two ways: *computation-based* and *memory-based*. The computation-based index function calculates the index in the kernel when the thread accesses the corresponding data element. The indices used in the memory-based index function have been calculated in the host side (*i.e.*, the CPU) before the kernel launches. The indices are stored in the shared memory when the kernel starts execution and are used during the kernel computation.

Exploiting vector I/O instructions. DeepCuts tries to exploit multi-word shared memory load operations (*e.g.*, 128-bit shared memory load). Although these operations do not reduce the shared memory latency, we observe that the performance increases because of the reduced number of load instructions.

Bank-conflict aware padding. When loading the input data into the shared memory, we observe that the shared memory's unaligned data layout results in heavy bank conflicts. To avoid this, DeepCuts applies padding to the input data to make them aligned.

Chapter 4

Optimization Framework for a GPU Cluster

4.1 Design choices of BigCuts

In this section, we introduce the key ideas of BigCuts, and the important design choices that affect the performance.

4.1.1 Storage-Based Training

Figure 4.1 describes the conventional approach of DNN training. For each training step, a portion of the training data is transferred from the storage to the GPU memory, whereas all other data and tensors (*e.g.*, parameter tensor, activation tensor, and gradient tensor) reside in GPU memory during the entire training process. In this case, the size of the GPU memory should be sufficiently large to hold all the tensors, which is impossible for huge transformers. For instance, it is known that we cannot train the GPT-2 model with 1.5B parameters[40] using conventional approach on a single V100 GPU with 32

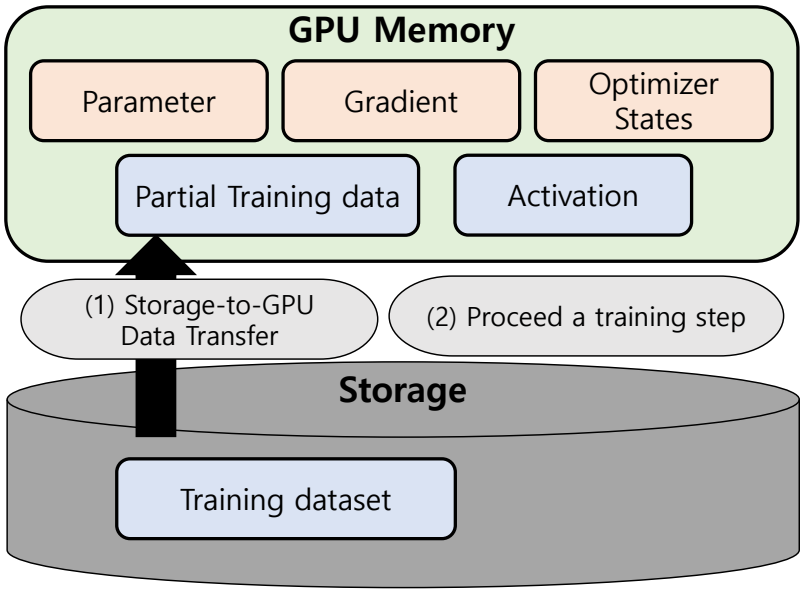


Figure 4.1: Conventional DNN training approach.

GB memory to the out of memory (OOM) error. In addition, even when we use a huge cluster system with 512 V100 GPUs, it is known that we cannot train transformer model that has more than 1 trillion parameters owning to the occurrence of an OOM error [23].

We can solve this problem using storage-based training. Figure 4.2 describes the approach of storage-based training. Storage-based training assumes that all the data and tensors are stored in the storage by default. We treat the training process as a sequence of partial steps (*e.g.*, a forward computation for the first layer). Before the execution of each step, all tensors required for the step are transferred from the storage to the GPU memory. Similarly, after the execution, the output tensors are transferred to the storage. Using this approach, we can train much larger DNN models on the same system than those of the conventional approach.

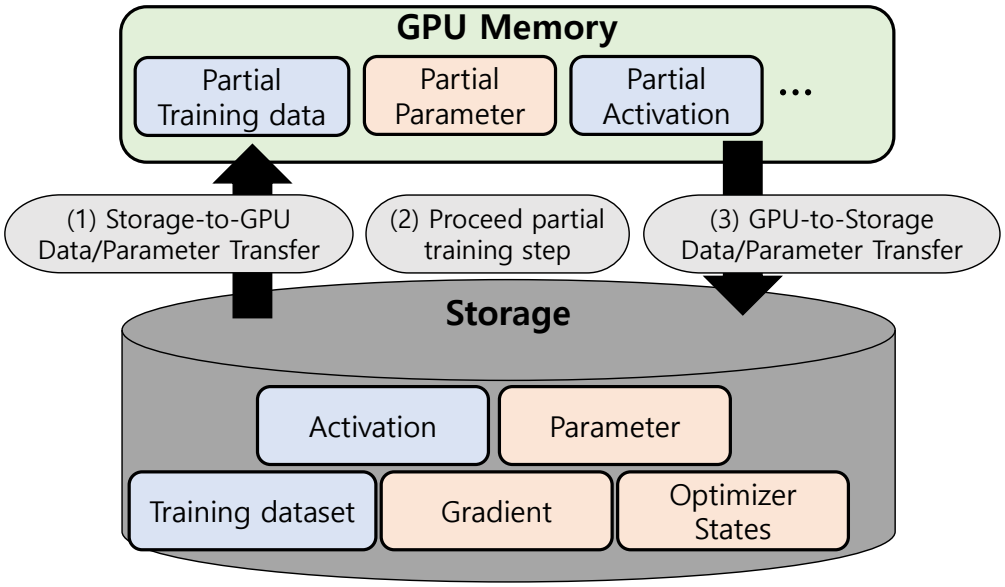


Figure 4.2: Storage-based DNN training approach.

To obtain a good performance for storage-based training, we need to minimize the overhead of a data transfer. The overhead of a data transfer depends on three factors: the transfer bandwidth determined by the hardware (*e.g.*, type of PCI-e and type of storage), the amount of data to be transferred, and the dependence between the transferred data and the computation (*i.e.*, whether the data transfer can be overlapped with the computation). Assuming that all data transfers can be overlapped with the computations (as discussed in the following subsection), we focus on reducing the amount of transferred data during the entire training step. The following sections describe the design choices that affect the amount of transferred data.

4.1.2 Workload Scheduling

Stages of DNN training DNN training has three stages: *forward propagation*, *backward propagation*, and *optimization*. During the forward propagation

stage, we compute the output of each layer (*i.e.*, activation tensors) from the first layer to the last layer. During the backward propagation stage, the computation is performed in the reverse direction; from the last layer to the first layer, we compute the gradients of the parameters. During the optimization stage, we update the parameters using the gradient and the optimization states (*e.g.*, momentum).

For the following sections, we focus on the forward propagation stage and the backward propagation stage, rather than the optimization stage, for two reasons. First, implementation of the optimization stage is very straightforward and has little room for optimization. For instance, the forward and backward propagations are compute-intensive for most DNN models. Therefore, the overhead of the data transfer can be hidden by overlapping it with the computations. For instance, we can upload the parameter for the second layer while applying forward propagation of the first layer. However, the workload of the optimization stage is memory-intensive. Thus, hiding the data transfer overhead is almost impossible, and the performance is bounded by the transfer bandwidth. As the other reason, the overhead of the optimization stage is relatively smaller than those of the other two stages.

Workload division To perform storage-based training, we divide the forward and the backward propagation into multiple partial steps. Figure 4.3 describes how the computations of the forward propagation are divided. There are two dimensions for the division: a division of the input data and a division of the parameters. In the case of the input data, a mini-batch of the input data is divided into multiple chunks, and each chunk is processed separately. We call these chunks micro-batches. Because the parameter is not updated until the entire mini-batch is processed, the same parameters can be used to process multiple

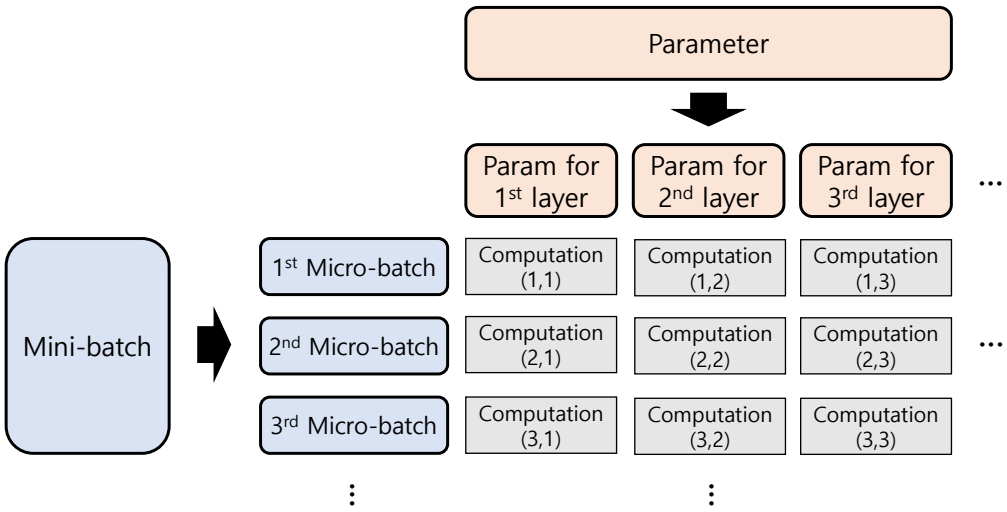


Figure 4.3: Division of computations of DNN training.

micro-batches. In the parameter dimensions, we can divide the computations in a layer-wise manner.

Scheduling policies The scheduling policy of the partial computations (*i.e.*, determining the order of executions) is an important factor for the performance of storage-based training because the order of execution greatly affects the overhead of data movement.

There are two policies of scheduling: *parameter-resident scheduling* and *activation-resident scheduling*. In the case of parameter-resident scheduling, the computations that share the same parameters are scheduled in adjacent time periods. For instance, in the case of Figure 4.3, the computations of the first layer (*e.g.*, (1,1), (2,1), and (3,1)) are executed first, and the computations for the second layer (*e.g.*, (1,2), (2,2), and (2,3)) are performed afterward. By doing this, we can reduce the overhead of the data movement of the parameters because the parameters that move to the GPU memory for the formerly exe-

cuted computations (*e.g.*, (1,1)) can be reused for the following computations (*e.g.*, (2,1) and (3,1))). Meanwhile, the activation tensors of each layer should be moved to storage because the distance between the two computations that are related to the same intermediate activation tensor (*e.g.*, (1,1) and (1,2)) is too large.

In the case of activation-resident scheduling, the computations related to a same micro-batch are scheduled in adjacent time periods (*e.g.*, (1,1), (1,2) and (1,3)). By doing this, the successive layer can directly use the output of the preceding layer without a data transfer overhead. However, the parameter of each layer should be moved from the storage to GPU memory for every iteration under this case.

4.1.3 Workload Distribution

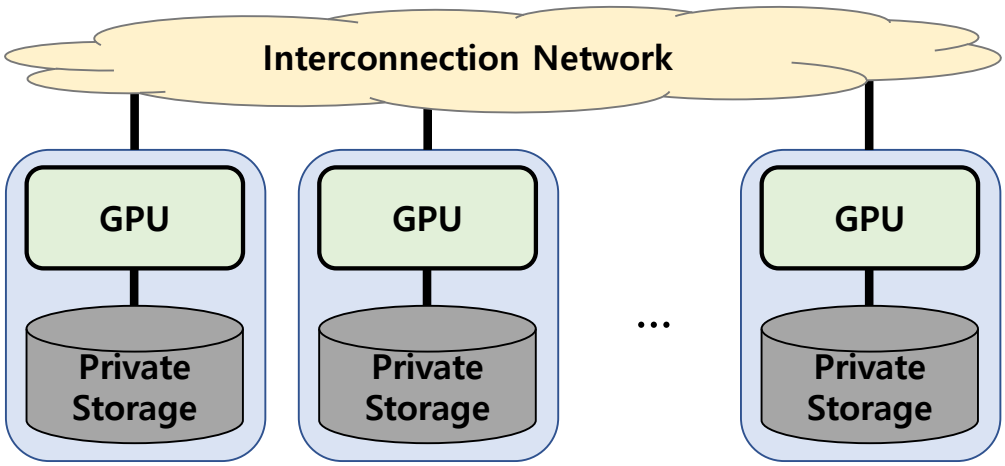


Figure 4.4: Abstraction of a GPU cluster system with storage.

Figure 4.4 describes the abstraction of the hardware system. BigCuts assumes that each GPU has private storage, and treats a pair of GPUs and the private storage (GPU-storage unit hereby) as a basic unit of workload distri-

bution. BigCuts assume that one MPI process is mapped to a GPU-storage unit.

BigCuts exploits three types of parallelism: data parallelism, intra-layer model parallelism (*i.e.*, weight partitioning), and inter-layer model parallelism (*i.e.*, pipelining). When exploiting data parallelism, each GPU-storage unit takes care of different micro-batches. For instance, the computations in a row, as shown in Figure 4.3 (*e.g.*, (1,1), (1,2), and (1,3)) are mapped to a single unit. When exploiting inter-layer model parallelism, computations in a column (*e.g.*, (1,1), (2,1), and (3,1)) are mapped to a single unit. In the case of intra-layer model parallelism, multiple GPU-storage units collaboratively process a single computation.

BigCuts can use multiple types of parallelism collaboratively. Depending on the way of parallelization, (1) the amount of the computation per GPU, (2) the amount of transferred data between GPUs, (3) the amount of data transferred between the GPUs and the private storage, and (4) the amount of storage usage for each private storage are determined. Each of these directly affects the throughput of the training. Thus, BigCuts use an analytical model that computes each factors, estimates the throughput using the performance model, and finds the best way of workload distribution. The detailed process is described in section 4.2.

4.1.4 Target workload

Target model In this thesis, we mainly target for the training workload of OOM-scale transformer models, because the sizes of the other type of models have not yet reached to the out-of-GPU-memory scale. In particular, we focus on the training workload of GPT-3[8]. However, the idea of the BigCuts can generally be applicable to other types of models as well.

Target precision BigCuts performs mixed-precision training[41]. For the forward propagation and the backward propagation stages, the parameters, the gradient of parameters, and the intermediate computation results (*i.e.*, activations and the gradient of activations) are represented using 16-bit floating-point values. For the optimization stage, parameters, gradients, and optimization states are represented using 32-bit floating-point values.

4.2 Performance Estimation Model

The goal of the performance estimation model of BigCuts is to find the best-performing way of the workload scheduling and workload distribution for the target training workload and target cluster system. To do so, we express them as a set of implementation parameters, and estimate the performance for each set of implementation parameters using an analytical model.

The analytical model is based on the following assumptions:

- The execution time largely depends on the parameterized complex operators (*e.g.*, convolutions and matrix multiplications in a dense layer).
- The data transfer for the forward and backward propagation can overlap with an independent GPU computation.
- If the execution time for the data transfer is shorter than the execution times of the GPU computations, we can completely hide the overhead of the data movement, as an ideal case. In this case, the total execution time depends on the computational efficiency.
- If the execution time for the data transfer is larger than the execution time of the GPU computation, we can completely hide the overheads of

the computations. In this case, the total execution time depends on the amount of data movement.

- For simplicity, we assume that the mini-batch size is a multiple of the micro-batch size.

The following sections describe the parameters and formula for computing the expected execution time.

4.2.1 Parameters

Table 4.1: Cluster Architecture Parameters

Symbol	Parameters
NUM_{GPU}	Number of GPUs in a cluster.
$PERF_{peak}$	Peak performance of half-precision floating-point operations
$BW_{storage}$	Bandwidth of the storage-to-GPU data transfer in a GPU-storage unit (GB/s)
BW_{P2P}	Bandwidth of GPU-to-GPU data transfer (GB/s)

Table 4.2: Parameters Obtained from Model Structure and Hyper-parameters

Symbol	Parameters
NUM_{ACT}	The average number of output neurons of a complex operator when the batch size is 1
NUM_{PARAM}	The average number of parameters for a complex operator
$NUM_{COMPUTE}$	The average number of floating point operations of a complex operator when the batch size is 1
$EFF_{COMPUTE}$	The expected computational efficiency of complex operators
NUM_{OP}	The total number of parameterized complex operators appears in a single pair of the forward and the backward propagation stage
$COEF_{OPTIMIZER}$	The number of optimizer-related values per a parameter value (*e.g.*, 4 for the ADAM optimzier)
$BATCH_{MINI}$	Mini-batch size

Table 4.3: Implementation Parameters

Symbol	Parameters
$Policy_{schedule}$	The scheduling policy. Zero if the parameter-resident scheduling is used, and 1 if the activation-resident scheduling is used
DIM_{DP}	The number of processes that processes different input data using same parameters
DIM_{WP}	The number of processes that cooperatively process a single layer computation
DIM_P	The number of processes involved in a pipeline
$BATCH_{MICRO}$	Micro-batch size

The parameters considered in BigCuts are described in Table 4.1, 4.2, and 4.3. Table 4.1 lists the parameters that describes the hardware characteristics of the target cluster. As described in section 4.1.3, we abstract the cluster as a set of the GPU-storage unit. For instance, in an NVIDIA DGX-2 node[35], 16 GPUs and 8 NVMe SSDs are installed. In this case, two GPUs share one NVMe SSD, and the $BW_{storage}$ is the half of the full bandwidth of a single NVMe SSD.

4.2.2 Performance Estimation

We start from the computations of the amount of data movement between the GPU and the private storage for a single parameterized complex operator. Assuming that we need to fetch all the data from storage, the amount of data movement in bytes is computed as follows. Note that each parameter value and the activation are represented as two bytes.

$$(NUM_{PARAM}/DIM_{WP} + NUM_{ACT} \cdot (BATCH_{MICRO}/DIM_{DP}) \cdot 2) \cdot 2$$

However, as described in section 4.1.2, the data in GPU memory can be reused without additional movement depending on the workload schedule. To

see this effect, We compute the total amount of data movement during the forward and backward propagation of a mini-batch. We call this $COMM_{FWDBWD}$. If we use parameter-resident scheduling, the amount of data movement is as follows.

$$COMM_{FWDBWD} = 4 \cdot NUM_{ACT} \cdot NUM_{OP} \cdot BATCH_{MINI}/(DIM_{DP} \cdot DIM_P)$$

If we use activation-resident scheduling, the amount of data movement is as follows:

$$COMM_{FWDBWD} = 2 \cdot NUM_{PARAM} \cdot NUM_{OP} \cdot BATCH_{MINI}/(DIM_{WP} \cdot DIM_P)$$

We also need to consider the overhead of data transfer for the optimization stage. The amount of data movement for the optimization stage, $COMM_{OPT}$, is computed as follows:

$$COMM_{OPT} = 8 \cdot COEF_{OPTIMIZER} \cdot NUM_{PARAM} \cdot NUM_{OP}/(DIM_{WP} \cdot DIM_P)$$

The total amount of data movement from and to storage, $COMM_{STORAGE}$, is summation of $COMM_{FWDBWD}$ and $COMM_{OPT}$.

$$COMM_{STORAGE} = COMM_{FWDBWD} + COMM_{OPT}$$

Meanwhile, we can compute the amount of data transferred between GPUs in a similar manner. We call the total amount of data transferred from/to other GPUs during the forward and the backward propagation of a mini-batch as $COMM_{GPU}$. This can be computed as follows:

$$COMM_{WP} = 2 \cdot ((NUM_{OP}/DIM_P) \cdot ((NUM_{ACT}/DIM_{DP}) \cdot log_2(DIM_{WP})) \cdot BATCH_{MINI}$$

$$COMM_{DP} = 2 \cdot (NUM_{PARAM}/DIM_{WP}) \cdot log_2(DIM_{DP})$$

$$COMM_P = 2 \cdot ((NUM_{ACT}/DIM_{DP}) \cdot BATCH_{MINI}$$

$$COMM_{GPU} = COMM_{WP} + COMM_{DP} + COMM_P$$

We then compute the expected execution time for the data transfer $TIME_{COMM}$ as follows:

$$TIME_{COMM} = COMM_{STORAGE}/BW_{STORAGE} + COMM_{DP} + COMM_P$$

We compare the expected execution time for the data transfer with the expected time for the computation, $TIME_{COMPUTE}$, which is computed as follows:

$$Computation = (NUM_{COMPUTE} \cdot NUM_{OP} \cdot BATCH_{MINI})$$

$$Performance = (EFF_{COMPUTE} \cdot PERF_{peak} \cdot DIM_{DP} \cdot DIM_{WP} \cdot DIM_P)$$

$$TIME_{COMPUTE} = Computation/Performance$$

As mentioned before, we assume that the computation and communication can fully overlap. Therefore, we can compute the expected execution time $TIME_{TOT}$ as follows:

$$TIME_{TOT} = Max(TIME_{COMPUTE}, TIME_{COMM})$$

Chapter 5

Evaluation

5.1 Optimization Framework for a Single GPU

In this section, we evaluate DeepCuts by comparing it with cuDNN and other state-of-the-art DNN optimization frameworks: TVM, TensorFlow-XLA, and TensorRT.

Table 5.1: System Configuration & Software Versions

CPU	2 x Intel Xeon Gold 6130 CPU (16-core, 2.1GHz)
GPU	NVIDIA Tesla V100, Volta architecture (5120 CUDA cores, 16GB HBM2) NVIDIA GeForce RTX 2080, Turing architecture (2944 CUDA cores, 8GB GDDR6)
Memory	256GB (DDR4 2400MHz)
Software	CUDA 11.2, cuDNN 8.0.5, PyTorch 1.7.1, TVM 0.8.dev, TensorFlow 1.14.0 and 2.4.1, TensorRT 7.2.2

5.1.1 Evaluation Environment for DeepCuts

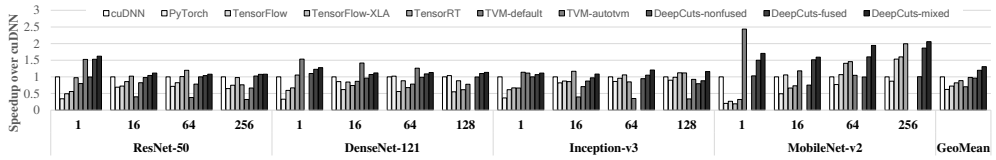
To check if DeepCuts and its performance model generically work for different GPU architectures, we evaluate DeepCuts on two different GPU architectures: NVIDIA Volta (V100) and Turing (RTX 2080). The system configuration and software tools used in the evaluation and their versionsis summarized in Table 5.1).

Table 5.2: Deep Learning Benchmark Applications

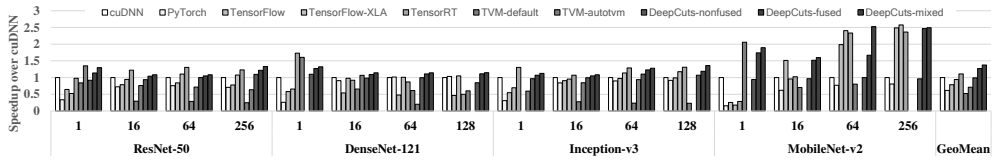
	Architecture	Dataset
CNN	ResNet-50[42]	ImageNet Dataset[43]
	DenseNet-121[44]	
	Inception-v3[45]	
	MobileNetV2[46]	
RNN	DeepSpeech2[3]	AN4[47]
	Attention-RNN[48]	WMT'17[49]
MLP	Facebook-DLRM[50]	Kaggle Display Advertising Challenge Dataset [51]
	BERT[4]	SQUAD 1.1[52]

Benchmark applications. Benchmark applications for the evaluation are summarized in Table 5.2. We denote transformer-based MLPs as MLPs hereafter.

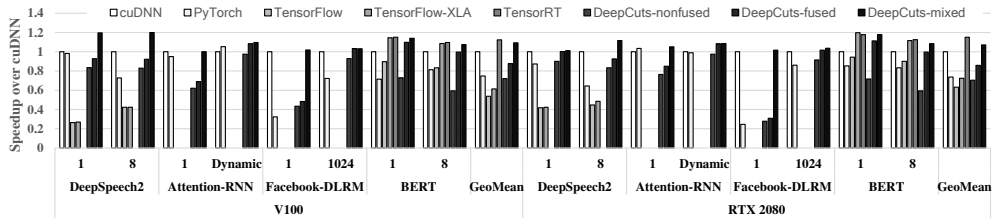
Measurements. We measure the performance of both training and inference workloads for each benchmark. For non-CNN benchmarks, we measure the execution time for one training epoch. For CNNs whose execution is very regular, we measure the training time for 100 mini-batches. We exclude the overheads for the file I/O and CPU-GPU data transfers (*e.g.*, loading initial parameters, copying input data into GPU memory, etc.) from the measurement because



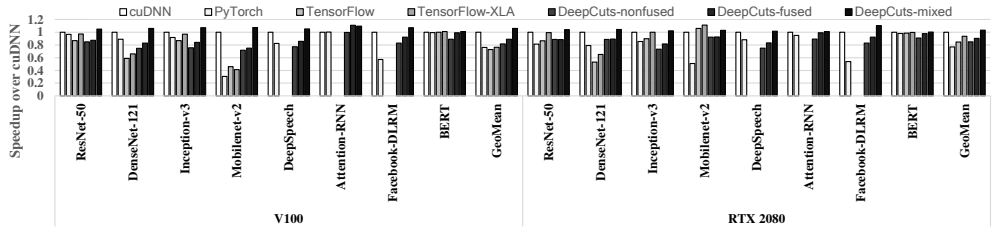
(a) CNN inference on V100.



(b) CNN inference on RTX 2080.



(c) RNN/MLP inference on V100 and RTX 2080.



(d) Training on V100 and RTX 2080.

Figure 5.1: Speedup over cuDNN for inference and training.

optimizing these are beyond the scope of this paper.

For each CNN model, we measure the performance of one training workload and four inference workloads with different batch sizes. For each non-CNN model, one training workload and two inference workloads with different batch sizes are evaluated. In total, 32 types (20 for CNNs, and 12 for non-CNNs) of DL workloads are used for evaluation.

DeepCuts versions. We evaluate three different versions of DeepCuts: DeepCuts-no-fusion, DeepCuts-fusion, and DeepCuts-mixed. DeepCuts-no-fusion uses kernels generated by DeepCuts without applying fusion. It generates kernels for both forward computation and backward computation. DeepCuts-fusion generates fused kernels. DeepCuts-mixed considers both cuDNN/cuBLAS primitives and kernels generated by DeepCuts-fusion when selecting which kernel to use.

Frameworks to compare. We compare DeepCuts with three state-of-the-art optimization frameworks: TVM (version 0.8.dev), TensorFlow XLA (version 2.4.1), and TensorRT (version 7.2.2).

We evaluate two versions of TVM: TVM-default (no autotuning capability) and TVM-autotvm. TVM-autotvm uses the CNN auto-tuning example script that provided in the official *autotvm* web site[53] .

Similar to TVM, we show two results of TensorFlow: TensorFlow and TensorFlow-XLA. TensorFlow-XLA is the case when the XLA optimization is applied to each workload. We do not show the results for Attention-RNN and Facebook-DLRM because we could not find reliable TensorFlow scripts for them. We use the official tutorial code[54, 55] for CNNs and DeepSpeech2. In the case of DeepSpeech2, we use TensorFlow 1.14 because the official tuto-

rial code for DeepSpeech2 is not working with TensorFlow 2.4.1. We use the reference code provided by NVIDIA[56] for BERT.

We use TensorRT-integrated TensorFlow[57] to evaluate TensorRT. For CNN inference, we use the same benchmark script used for TensorFlow and TensorFlow-XLA. For BERT, we use reference implementation provided by NVIDIA[58]. For other models, we do not find any publicly available TensorFlow scripts that work with TensorRT.

We also compare DeepCuts with a widely-used DL framework, PyTorch (version 1.7.1). For the CNN models, we use the official tutorial scripts of PyTorch[59]. For non-CNN models, we use scripts provided by the author of the original paper[60] or from some widely-used projects[61].

We observe that the runtime overhead from existing frameworks itself is not negligible for some benchmarks. For a model whose computational cost is relatively small, this overhead becomes significant. To get rid of this, we implement a manual C++ implementation that directly calls GPU kernels, denoted cuDNN. In cuDNN, the computation is performed using cuDNN/cuBLAS primitives and CUDA kernels with negligible runtime overheads. We use cuDNN as the baseline in the evaluation.

Table 5.3: # of Top-performing Workloads on V100

Model Type	Number of top-performing workloads				
	TensorFlow -XLA	TensorRT	TVM -default	TVM -autotvm	DeepCuts -mixed
CNN	1	1	3	2	13
RNN	0	0	0	0	6
MLP	0	2	0	0	4
Total	1	3	3	2	23

Table 5.4: # of Top-performing Workloads on RTX 2080

Model Type	Number of top-performing workloads				
	TensorFlow -XLA	TensorRT	TVM -default	TVM -autotvm	DeepCuts -mixed
CNN	1	5	1	1	12
RNN	0	0	0	0	6
MLP	0	2	0	0	4
Total	1	7	1	1	22

5.1.2 Evaluation Results

Overall performance. Table 5.3 summarizes the number of top-performing workloads for each framework on V100. Among the 32 workloads, DeepCuts-mixed shows the best performance for 23 workloads. It outperforms all other frameworks on average. For the rest nine workloads, TensorRT and TensorFlow-XLA shows slightly better performance than DeepCuts-mixed (less than 1.1×) for two large-batched CNN workloads. TVM-default is the best performer for three small-batched inferences. TVM-autotvm is the best performer for two DenseNet-121 inferences (batch sizes 16 and 64). TensorRT is the top-performer for two BERT inference workloads, because it uses several BERT-specialized kernels that are manually implemented.

DeepCuts-fused outperforms all other frameworks for nine workloads. This is a notable result that shows DeepCuts can outperform other frameworks without using cuDNN primitives. On the other hand, DeepCuts-nonfused is outperformed by at least one other framework for all the workloads. This is natural in that other frameworks (TensorFlow-XLA, TensorRT, and TVM) perform the fusion optimization while DeepCuts-nonfused does not.

The overall trends on RTX 2080 are similar to those on V100. DeepCuts-mixed is the top performer for 22 workloads and outperforms all other frame-

works on average. This indicates that the performance model of DeepCuts can be generically applicable to different GPU architectures.

Among all the DL operations in the 32 workloads, DeepCuts-mixed uses cuDNN/cuBLAS primitives only for 25% of the DL operations. For the remaining 75%, DeepCuts generates its own CUDA kernels.

CNN inference on V100. Figure 5.1(a) shows the performance of CNN inference on V100. The speedup is obtained over cuDNN. The batch sizes are 1, 16, 64, and 256. We use a batch size of 128 instead of 256 for some cases because of the GPU memory capacity. Some results of TVM-autotvm are omitted because TVM fails to tune at DenseNet-121 with the batch sizes of 1 and 128, and Inception-v3 with the batch size of 64. In addition, the result of TVM-default for MobileNet-v2 is also omitted because of out-of-memory error.

Overall, DeepCuts-mixed shows the best performance on average. In addition, DeepCuts-no-fusion achieves competitive performance to cuDNN (0.96×). After applying fusion, DeepCuts-fusion is 1.19× faster than cuDNN.

When the batch size is one, DeepCuts-mixed, DeepCuts-fusion, DeepCuts-no-fusion, TVM-default, and TVM-autotvm significantly outperforms cuDNN. There are two reasons for this. One is that cuDNN is not well-optimized for small-batch sizes while DeepCuts and TVM generate well-optimized kernels for these cases. The other is that DeepCuts and TVM fuse operations while cuDNN or cuBLAS cannot fuse them.

For the largest-batch inference (128 or 256 depending on applications), DeepCuts-mixed, DeepCuts-fusion outperforms cuDNN while DeepCuts-no-fusion, TVM-default, and TVM-autotvm perform worse than cuDNN on average. Even though DeepCuts-no-fusion is worse than cuDNN (0.95×), DeepCuts-fusion outperforms cuDNN (1.11×). TensorFlow-XLA and TensorRT also perform better

than cuDNN for the largest-batch CNN inference.

DeepCuts-mixed, DeepCuts-fusion, TensorFlow-XLA, and TensorRT are much better than cuDNN especially for MobileNet-v2 with a large batch size (64 or 256). This is because the inference workload of MobileNet-v2 is suitable for kernel fusion. The percent execution time of non-convolutional and memory-intensive operations in MobileNet-v2 is over 60% of the total execution time, while the percent execution time of memory-intensive operations in the other three benchmarks is less than 30%. By fusing these operations with compute-intensive convolution operations in MobileNet-v2, memory access overhead can be eliminated significantly.

PyTorch and TensorFlow are significantly slower than cuDNN on average. The overhead of interpreting python scripts becomes relatively large when handling light-weight small-batch kernels (*e.g.*, when the batch size is one). For the same reason, TensorFlow-XLA and TensorRT are worse than cuDNN when the batch size is one (0.46× and 0.68× respectively).

CNN inference on RTX 2080. Figure 5.1(b) shows the evaluation results on RTX 2080. Overall, the trends on RTX 2080 are similar to those on V100. DeepCuts-mixed outperforms cuDNN, TensorFlow-XLA, TensorRT, and TVM-autotvm on average. This indicates that the performance model of DeepCuts can be generically applicable to different GPU architectures.

For MobileNet-v2, the speedups of DeepCuts-mixed and DeepCuts-fusion over cuDNN on RTX 2080 are much higher (2.08× and 1.81×) than those on V100 (1.81× and 1.61×). This is because the global memory bandwidth of RTX 2080 (GDDR6, 448 GB/s) is smaller than that of V100 (HBM2, 900 GB/s). Thus, the effect of fusion that reduces global memory accesses becomes significant for RTX 2080.

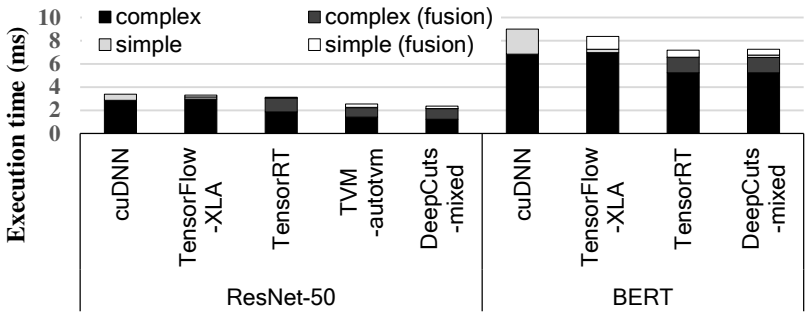


Figure 5.2: Exec. time breakdown for ResNet-50 and BERT.

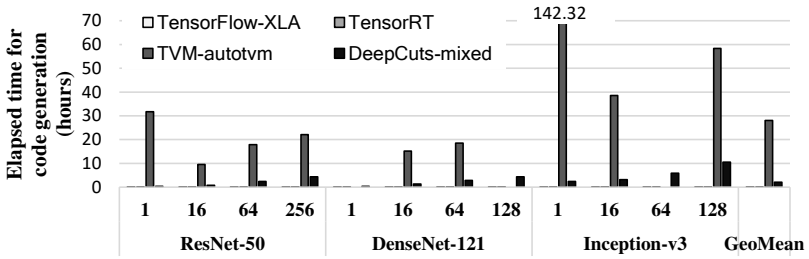


Figure 5.3: Elapsed time for code generation. The bars for the elapsed times of TensorFlow XLA and TensorRT are almost invisible because they take only a few seconds.

RNN/MLP inference. Figure 5.1(c) shows the result of RNN/MLP inference on V100 and RTX 2080. For Attention-RNN, the batch size varies significantly depending on the lengths of the input sequences in the batch. We do not show the performance of TVM because TVM does not support some important DL operations of RNNs/MLPs, such as bidirectional GRU and embedding.

DeepCuts-mixed is 1.05× faster than cuDNN on average. For small batches in Attention-RNN and Facebook-DLRM, DeepCuts-fusion and DeepCuts-no-fusion are worse than cuDNN. These applications heavily rely on matrix-matrix multiplication that is implemented using cuBLAS in cuDNN. The matrix-matrix multiplication becomes a matrix-vector multiplication when the batch size is one. We observe that DeepCuts fails to achieve near-cuBLAS performance for this case. For small-batch BERT inference, TensorRT outperforms DeepCuts-mixed and cuDNN because of its manually-tuned kernels.

Training workloads. Figure 5.1(d) shows the performance of training. For each application, we use the maximum batch size allowed by the GPU memory size. On average, DeepCuts-mixed is 1.04× faster than cuDNN while DeepCuts-fusion and DeepCuts-no-fusion is slower than cuDNN. The performance of DeepCuts-fusion and that of DeepCuts-no-fusion are almost the same. This indicates that the performance gain obtained by fusion is relatively small for the training workload compared to the inference workload. DeepCuts-mixed is faster than cuDNN because DeepCuts generates kernels that are faster than cuDNN for 16% of DL operations. TensorFlow-XLA worse than cuDNN on average and rarely achieves speedup over cuDNN for all applications.

5.1.3 Execution Time Breakdown

Figure 5.2 shows the execution time breakdown of GPU kernels for ResNet-50 and BERT inference measured with the NVIDIA nvprof profiler[38]. The batch size is one. We categorize the GPU kernels into four types; complex, complex (fusion), simple, and simple (fusion). The definition of complex and simple is the same as described in Table 3.1. complex (fusion) represents the fused kernels that include at least one complex operation. simple (fusion) represents the fused kernels that consist of simple operations only. The overheads that are not related to the GPU kernel computation (*e.g.*, CPU-GPU data transfer overheads, CPU overheads, etc.) are not included in Figure 5.2.

For both benchmarks, the total execution time of complex operations (the sum of complex and complex (fusion)) of DeepCuts-mixed is smaller than that of cuDNN (0.75× and 0.95× for ResNet-50 and BERT, respectively). This is because convolution and matrix multiplication kernels generated by DeepCuts outperform cuDNN/cuBLAS primitives for some DL operations. In addition, the execution time of simple operations (the sum of simple and simple (fusion)) is greatly reduced (0.4× and 0.33×) because of the kernel fusion optimization.

The execution time breakdown for DeepCuts-mixed is quite similar to that of the well-performing competitor for each benchmark (TVM-autotvm for ResNet-50, and TensorRT for BERT). This is quite interesting because DeepCuts' optimization methodology is entirely different from other frameworks, *i.e.*, machine-learning-based estimation for TVM-autotvm, manually-tuned kernels for TensorRT, and simple-model-based estimation for DeepCuts-mixed.

5.1.4 Code Generation Time

Figure 5.3 compares the code generation time between DeepCuts-mixed, TVM-autotvm, TensorFlow-XLA, and TensorRT. It is hard to find a pattern for the code generation time of TVM. We observe that, even when TVM generates code for the same DNN model, the code generation time and the code quality vary for every instance of code generation. It seems that the randomness in the *autotvm*'s simulated annealing process significantly affects the code generation time and quality. To handle the unstable code generation time of *autotvm*, we generate code for each configuration of TVM-autotvm twice and pick the result with better performance.

TensorFlow-XLA and TensorRT generate code much faster than DeepCuts (1,058× and 652×). While DeepCuts spends a lot of time on finding the best-performing implementation parameters of convolution and matrix multiplication, TensorFlow-XLA just uses cuDNN/cuBLAS primitives and TensorRT uses pre-implemented kernels resulting in much faster code generation time. On the other hand, DeepCuts-mixed generates optimized code 13.53× faster than TVM-autotvm. The code generation time of DeepCuts is almost linearly proportional to the search space size that is, in turn, proportional to $log(B)$ where B is the batch size.

5.2 Optimization Framework for a GPU Cluster

5.2.1 Evaluation Environment

Table 5.5 describes the target cluster systems. Unlike previous studies [23, 21] that are optimized for a specific system (DGX-2 and DGX SuperPOD) provided by NVIDIA, BigCuts targets for verstaile types of cluster systems including. Cluster A and Cluster C are the systems that are built with COTS components.

Table 5.5: Target Cluster Systems

	Cluster A	Cluster B	Cluster C
Number of nodes	8	2	4
CPU	AMD EPYC 7502P	Intel Xeon Platinum 8168	AMD EPYC 7502P
GPUs	4 × NVIDIA RTX 3090	8 × NVIDIA V100	4 × NVIDIA RTX 3090
Intra-node GPU-to-GPU connection	PCIe Gen4	12 × NVSwitch	PCIe Gen4
Storage per node	80 × HDDs with 2 × raid cards	8 × NVMe SSDs	1 × NVMe SSDs
per-GPU peak performance (FP16)	285.48 TFLOPS	125 TFLOPS	285.48 TFLOPS
Intra-node Comm. BW (GPU to GPU)	16 GB/s	50 GB/s	16 GB/s
Inter-node Comm. BW (Node to Node)	25 GB/s	100 GB/s	25 GB/s
Storage-to-GPU Comm. BW	4 GB/s per GPU	1.6 GB/s per GPU	0.4 GB/s per GPU

Cluster B is a DGX SuperPOD system.

5.2.2 Evaluation Results

Performance estimation In this section, we show the importance of adapting the design choices for the given hardware architecture and the DNN model. We show the estimation result using two different out-of-GPU-memory-scale models, GPT-3[8] and Turing-NLG[62]. We use the same mini-batch size and the optimizer that is used in the original papers. We use two different cluster architectures, Cluster A and Cluster B, described in Table 5.5.

Table 5.6 shows the expected execution time varying the way of parallelization and the way of workload scheduling. We normalize the execution time to the best-case execution time. DP, WP, and P represent the number of data parallel groups, the number of weight partitioning groups, and the num-

ber of pipelining groups. PR, AR represent parameter-resident scheduling and activation-resident scheduling. We do not use more than eight pipeline groups for Turing-NLG, because the mini-batch size is too small to perform pipelining.

We can observe that the optimal design choices (*i.e.*, DP, WP, P and the choice of scheduling) varies depending on the hardware architecture and the model structure. In other words, there are no trivial solutions for selecting the design choice. This result shows why we need to use the estimation model; to find the best-performing design choices before the execution, an estimation model is required.

We also compared the estimated performance with the ideal-case estimated performance (*i.e.*, the expected performance when the computations are performed without any communication). In the case of GPT-3 training, the best-case execution time is same as that of ideal-case execution time on both cluster systems. We can observe that BigCuts performance model finds several well-performing configurations that the communication overhead can be hidden by the overhead of the computations.

However, in the case of Turing-NLG, the best-case execution time is 4.25× and 3.61× longer than the ideal cases on Cluster A and Cluster B, respectively. The main reason for this is the mini-batch size. The mini-batch size of GPT-3 training is 320,000, whereas that of the Turing-NLG training is 512. The small mini-batch size results in three problems. The first one is the overhead of the gradient reduction across the data-parallel dimension. Because the gradient should be reduced before the weight update, the overhead of GPU-to-GPU for this (denoted as $COMM_{DP}$ in section 4.2) increases. The second problem is the overhead of the optimization. For each mini-batch, the tensors for the optimization stage should be loaded from storage, which results in a huge overhead. As the last problem, the pipeline becomes inefficient for small mini-batch training.

After a mini-batch of data are trained, the pipeline should be filled from empty again because the parameter is changed.

Implementation We implement the prototype of BigCuts. The prototype of BigCuts perform inference workload for OOGM-scale transformer models. We perform evaluation using GPT3-style OOGM model with 6.7B parameters (*i.e.*, GPT-3 6.7B in the original paper[8]). We used Cluster-C system for our evaluation. The size of the micro-batch is 8.

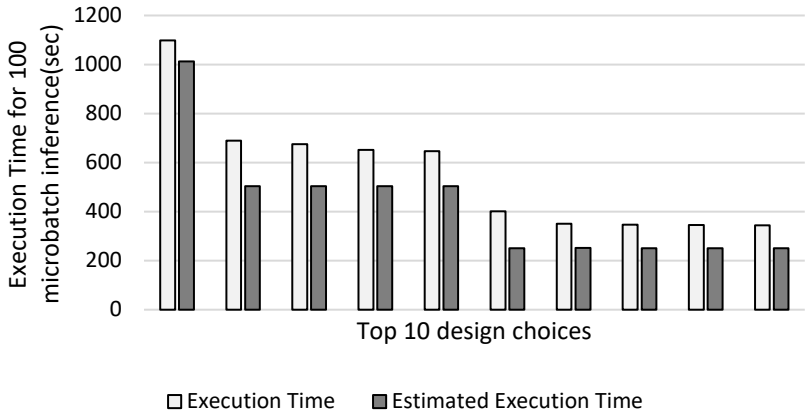


Figure 5.4: Execution time of GPT-3 6.7B inference for 100 micro-batches.

Figure 5.4 shows the execution time of BigCuts prototype. We show the execution time for the top-10 configurations. Overall, the execution time for inference is determined by the storage bandwidth. Since the model estimates the overhead of the storage accesses pretty well, the model successfully finds the optimal design choices.

We observe that, on Cluster C, the GPT-3 training performance is bounded by the storage bandwidth, because each node has only one NVMe SSD. Even for the best-performing configuration (two data-parallel groups, eight pipeline stages, and parameter-resident scheduling), the execution time for a storage-

GPU data transfer is 4.05× longer than that of a GPU computation.

The measured execution time is longer than the estimated execution time. There are two reasons for this. The first reason is the storage bandwidth. The model uses the storage bandwidth measured from the micro-benchmark. However, we observed that the bandwidth achived during the inference is slightly lower (less than 10% degradation) than the micro-benchmark execution. We observed that the handling of small tensors *(e.g.*, parameters for non-matrix multiplication operations), which is negligible in the model, affects the performance and results in bandwidth degradation. The other problem is the overlapping. For simplicity of the model, we assume that the overhead for the computation can be fully hidden by the overhead of the communication when the overhead of communication is huge enough. Hence, they are not fully hidden because the prototype of BigCuts is not fully optimized yet. We will solve this problem in a future study.

Note that, even if the estimated execution time is not exactly the same as the real execution time, the model is still useful for finding the optimal design choices because the overall trends are quite similar. We can use the model to select candidate design choices and can choose the best-performing model by brute-forcely trying a few design choices.

Table 5.6: Normalized Execution Time to the Ideal Case

	GPT-3				Turing-NLG			
	Cluster A		Cluster B		Cluster A		Cluster B	
(DP, WP, P)	PR	AR	PR	AR	PR	AR	PR	AR
(32, 1, 1)	95.17	1.01	22.48	3.72	22.25	3.02	3.93	8.3
(16, 1, 2)	47.59	1.02	11.25	3.69	11.13	1.68	1.98	4.6
(16, 2, 1)	48.58	2.24	14.9	7.41	11.45	2.08	2.86	5.5
(8, 4, 1)	27.77	4.97	20.28	14.89	6.86	2.3	4.56	5.48
(8, 2, 2)	24.79	2.23	9.31	7.39	5.89	1.41	1.89	3.65
(8, 1, 4)	23.8	1.01	5.66	3.67	5.57	1.07	**1**	2.75
(4, 8, 1)	23.81	10.91	46.75	29.97	6.68	3.91	11.22	8.23
(4, 4, 2)	15.88	4.97	17.51	14.87	4.09	1.96	4.08	4.55
(4, 2, 4)	12.91	2.23	6.54	7.38	3.12	**1**	1.41	2.73
(4, 1, 8)	11.91	**1**	2.89	3.67	N/A	N/A	N/A	N/A
(2, 16, 1)	37.71	23.8	118.56	60.38	11.79	7.96	28.83	15.19
(2, 8, 2)	17.89	10.91	45.43	29.95	5.3	3.74	10.99	7.77
(2, 4, 4)	9.96	4.97	16.18	14.87	2.71	1.8	3.86	4.09
(2, 2, 8)	6.98	2.24	5.21	7.38	N/A	N/A	N/A	N/A
(2, 1, 16)	5.99	1	1.56	3.66	N/A	N/A	N/A	N/A
(1, 32, 1)	82.36	51.57	293.51	121.69	26.68	16.96	71.57	29.91
(1, 16, 2)	34.78	23.81	118	60.38	11.11	7.88	28.75	14.96
(1, 8, 4)	14.96	10.92	44.89	29.97	4.62	3.66	10.92	7.54
(1, 4, 8)	7.02	4.98	15.64	14.87	N/A	N/A	N/A	N/A
(1, 2, 16)	4.04	2.26	4.66	7.38	N/A	N/A	N/A	N/A
(1, 1, 32)	3.02	1.49	**1**	3.66	N/A	N/A	N/A	N/A

Chapter 6

Discussions and Future Work

Supporting versatile type of operations DeepCuts mainly focuses on supporting operations used in widely-used DNN models. However, thanks to DeepCuts' flexible code generation scheme, it is possible to generate efficient kernels for uncommon types of DL operations.

To show the flexibility of DeepCuts, we perform a preliminary study on two types of exotic operations: the Shift-Conv-Shift-Conv (SC^2) module[63] and Octave convolution[64]. SC^2 module includes two shift operations and two convolution operations in an alternative manner. A shift operation is a simple data movement operation that shifts the input pixels to their neighbors. We find that DeepCuts can generate efficient kernels for the SC^2 module only with a few modifications of the code generator. In this case, shift operations are fused with the following convolution operation, resulting in an efficient fused kernel.

Supporting Octave convolution is a bit harder. Octave convolution assumes that the channels are divided into two parts, a low-frequency part and a high-frequency part, and uses different feature map sizes between them. When per-

forming convolution, pooling or up-sampling is applied to the input channels before applying the usual convolution. We find that DeepCuts can support this by generating two different convolution kernels, one for the low-frequency part in the output and the other for the high-frequency part. However, generating a single fused kernel for Octave Convolution is not supported for the current version of DeepCuts. To do this, the performance model and the code generator of DeepCuts need be modified to support horizontal fusion[65]. This is included in our future work.

Supporting versatile GPU architectures The current version of DeepCuts supports NVIDIA GPUs only. We are currently working on the OpenCL back-end of DeepCuts to support versatile GPU architectures including AMD GPUs and mobile devices. Although there are several minor issues that are originated from the differences between OpenCL and CUDA (*e.g.*, OpenCL does not support atomic float instruction), we expect that the overall idea of DeepCuts would be applied well for other types of architectures because the performance model is based on the modern GPUs' standard architecture parameters.

BigCuts implementation In this thesis, we show the performance of BigCuts prototype for inference workloads only. Since the computation/communication pattern of inference is very regular, the performance model of BigCuts estimates the performance very well. For now, we are implementing BigCuts for training workloads, which is more complicated than the inference workload. We plan to evaluate it on a larger GPU cluster with a higher storage-GPU communication bandwidth.

Chapter 7

Conclusions

In this thesis, we propose performance-model-driven approaches to handle DL workloads' versatility and to handle the out-of-GPU-memory scale model training.

We propose DeepCuts, a DL optimization framework for a single GPU. For the DL operations set in a given DL workload, DeepCuts searches for best-performing implementation parameters for the DL operations considering their fusion and prunes definitely-slow cases of the implementation parameters using a performance estimation model. Then it generates GPU kernels using DFGs for the selected groups of implementation parameters, executes the kernels, and determines the best performing kernel for each DL operation or fused DL operation in the DL workload.

For DL workloads, including both training and inference of CNNs, RNNs, and transformer-based MLPs, the performance of DeepCuts is comparable to or better than cuDNN. When DeepCuts includes cuDNN primitives in its kernel selection process, it brings speedups of 1.13 and 1.46 over cuDNN and

PyTorch on the NVIDIA V100 GPU, respectively. When compared to TVM that is the state-of-the-art DL optimization framework but does not support training, DeepCuts achieves comparable performance to TVM with the code generation time reduced by 13.53×. It achieves the speedup of 1.24 over TVM when including cuDNN primitives in its kernel selection process. Compared to TensorFlow-XLA that supports training, DeepCuts achieves a speedup of 1.38. We show that DeepCuts also achieves good performance on the NVIDIA RTX 2080 GPU with a significantly different architecture to NVIDIA V100.

In addition, we propose BigCuts, a DL optimization framework for out-of-GPU-memory scale model training on a GPU cluster. BigCuts enables the out-of-GPU-memory scale training on a relatively smaller cluster than other approaches. Thanks to the performance estimation model, BigCuts successfully finds the best-performing way of workload distribution and scheduling for the given training workload and hardware architecture.

Bibliography

[1] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[2] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016.

[3] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International conference on machine learning*, pp. 173–182, 2016.

[4] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner,

L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," *CoRR*, vol. abs/1912.01703, 2019.

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *CoRR*, vol. abs/2005.14165, 2020.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[10] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015.

[11] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *CoRR*, vol. abs/2101.03961, 2021.

[12] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014.

[13] C. Leary and T. Wang, "Xla: Tensorflow, compiled," *TensorFlow Dev Summit*, 2017. Accessed: 2021-05-10.

[14] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

[15] NVIDIA, "Nvidia tensorrt." `https://developer.nvidia.com/tensorrt`. Accessed: 2020-05-11.

[16] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, R. Kimball, J. Knight, N. Korovaiko, V. K. Vijay, Y. Lao, C. R. Lishka, J. Menon, J. Myers, S. A. Narayana, A. Procter, and T. J. Webb, "Intel ngraph: An intermediate representation, compiler, and executor for deep learning," *CoRR*, vol. abs/1801.08058, 2018.

[17] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *CoRR*, vol. abs/1802.04730, 2018.

[18] T. D. Le, H. Imai, Y. Negishi, and K. Kawachiya, "TFLMS: large model support in tensorflow by graph rewriting," *CoRR*, vol. abs/1807.02037, 2018.

[19] B. Salimi, L. Rodriguez, B. Howe, and D. Suciu, "Capuchin: Causal database repair for algorithmic fairness," *CoRR*, vol. abs/1902.08283, 2019.

[20] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. A. Hechtman, "Mesh-tensorflow: Deep learning for supercomputers," *CoRR*, vol. abs/1811.02084, 2018.

[21] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *CoRR*, vol. abs/1909.08053, 2019.

[22] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "Zero-offload: Democratizing billion-scale model training," *CoRR*, vol. abs/2101.06840, 2021.

[23] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the GPU memory wall for extreme scale deep learning," *CoRR*, vol. abs/2104.07857, 2021.

[24] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015.

[25] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 47–62, 2019.

[26] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pp. 38–38, IEEE, 1998.

[27] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi™*, pp. 167–188, Springer, 2014.

[28] B. van Werkhoven, "Kernel tuner: A search-optimizing gpu code auto-tuner," *Future Generation Computer Systems*, vol. 90, pp. 347–358, 2019.

[29] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 303–316, ACM, 2014.

[30] P. Pfaffe, T. Grosser, and M. Tillmann, "Efficient hierarchical online-autotuning: a case study on polyhedral accelerator mapping," in *Proceedings of the ACM International Conference on Supercomputing*, pp. 354–366, ACM, 2019.

[31] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 22, IEEE Press, 2016.

[32] W. Jung, D. Jung, S. Lee, W. Rhee, J. H. Ahn, *et al.*, "Restructuring batch normalization to accelerate cnn training," *arXiv preprint arXiv:1807.01702*, 2018.

[33] S. Li, "Getting started with distributed data parallel." https://pytorch.org/tutorials/intermediate/ddp_tutorial.html. Accessed: 2021-06-29.

[34] Google, "Distributed training with tensorflow." https://www.tensorflow.org/guide/distributed_training. Accessed: 2021-06-29.

[35] NVIDIA, "Dgx-2: Ai servers for solving complex ai challenges." https://www.nvidia.com/en-us/data-center/dgx-2/. Accessed: 2021-06-02.

[36] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," vol. abs/1409.1556, 2014.

[37] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009.

[38] NVIDIA, "Nvidia visual profiler." https://developer.nvidia.com/nvidia-visual-profiler, 2014.

[39] J. Cheng, M. Grossman, and T. McKercher, *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[40] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[41] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," *CoRR*, vol. abs/1710.03740, 2017.

[42] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[43] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[44] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[45] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[46] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," *CoRR*, vol. abs/1801.04381, 2018.

[47] "The cmu audio databases." `http://www.speech.cs.cmu.edu/databases/an4/index.html`. Accessed: 2019-4-15.

[48] M. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *CoRR*, vol. abs/1508.04025, 2015.

[49] "Emnlp 2017 second conference on machine translation (wmt17)." `http://www.statmt.org/wmt17/translation-task.html`. Accessed: 2019-4-15.

[50] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep learning recommendation model for personalization and recommendation systems," *CoRR*, vol. abs/1906.00091, 2019.

[51] "Display advertising challenge." `https://www.kaggle.com/c/criteo-display-ad-challenge/overview`. Accessed: 2019-4-15.

[52] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100, 000+ questions for machine comprehension of text," *CoRR*, vol. abs/1606.05250, 2016.

[53] L. Zheng and E. Yan, "Auto-tuning a convolutional network for nvidia gpu." `https://docs.tvm.ai/tutorials/autotvm/tune_relay_cuda.html`. Accessed: 2019-11-21.

[54] "tf_cnn_benchmarks: High performance benchmarks." `https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_CNN_benchmarks`. Accessed: 2019-11-21.

[55] "Tensorflow implementation of deepspeech2." `https://github.com/yao-matrix/deepSpeech2`. Accessed: 2019-11-21.

[56] "Bert for tensorflow." `https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/LanguageModeling/BERT`. Accessed: 2020-04-17.

[57] "Accelerating inference in tf-trt user guide." https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html. Accessed: 2020-04-14.

[58] "Bert example using the tensorrt c++ api." https://github.com/NVIDIA/TensorRT/tree/release/5.1/demo/BERT/. Accessed: 2020-04-14.

[59] S. Chintala, "Imagenet training in pytorch." https://github.com/pytorch/examples/tree/master/imagenet. Accessed: 2019-11-21.

[60] Facebook, "An implementation of a deep learning recommendation model (dlrm)." https://github.com/facebookresearch/dlrm. Accessed: 2019-11-21.

[61] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A fast, extensible toolkit for sequence modeling," in *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[62] Microsoft, "Turing-nlg: A 17-billion-parameter language model by microsoft." https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/. Accessed: 2021-06-02.

[63] B. Wu, A. Wan, X. Yue, P. Jin, S. Zhao, N. Golmant, A. Gholaminejad, J. Gonzalez, and K. Keutzer, "Shift: A zero flop, zero parameter alternative to spatial convolutions," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9127–9135, 2018.

[64] Y. Chen, H. Fan, B. Xu, Z. Yan, Y. Kalantidis, M. Rohrbach, Y. Shuicheng, and J. Feng, "Drop an octave: Reducing spatial redundancy in convolu-

tional neural networks with octave convolution," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 3434–3443, 2019.

[65] A. Li, B. Zheng, G. Pekhimenko, and F. Long, "Automatic horizontal fusion for GPU kernels," *CoRR*, vol. abs/2007.01277, 2020.

초록

PyTorch, TensorFlow와 같은 딥 러닝 프레임워크는 딥 러닝 소프트웨어 에코시스템에서 가장 중요한 역할을 하는 핵심적인 소프트웨어이다. 그러나 현존하는 딥 러닝 프레임워크들은 특정 종류의 딥 러닝 워크로드에만 맞추어 최적화되어있으며, 최근 새로이 개발된 딥 러닝 모델을 활용하고자 하는 경우 성능과 활용성이 크게 떨어진다. 크게 두 가지 원인이 있다. 첫 번째로, 현존하는 딥 러닝 프레임워크들은 구현이 고정된 라이브러리(예: cuDNN)을 사용하는데, 이 때문에 다양한 종류의 딥 러닝 모델에 대해 좋은 성능을 보이기 어렵다. 또한 GPU 메모리상에 필요한 데이터가 전부 저장되는 상황만을 주로 가정하고 구현되어 있어 학습하고자 하는 모델의 크기가 GPU 메모리에 올릴 수 없을 만큼 큰 경우 학습 코드를 작성하기 어렵다.

본 논문은 이를 해결하기 위한 솔루션으로 DeepCuts와 BigCuts를 제시한다. DeepCuts는 주어진 딥 러닝 워크로드와 GPU 하드웨어 구조를 동시에 고려하여 최적화된 커널 코드를 생성하는 프레임워크이다. DeepCuts는 딥 러닝 워크로드를 표현하는 연산 그래프(computation graph)를 입력으로 받아 최적화된 GPU 커널을 생성한다. 이 과정에서 여러 개의 딥 러닝 연산을 합성하는 커널 퓨전(Kernel fusion)을 적용하며, 커널의 성능 상한을 예측하는 성능 모델을 사용하여 커널 생성을 최적화한다. DeepCuts는 cuDNN/cuBLAS를 사용하는 경우보다 높은 성능을 보이며, TVM, TensorFlow XLA, TensorRT등의 세계 최고 수준 딥 러닝 최적화 프레임워크 및 컴파일러와 비슷하거나 더 높은 성능을 보인다.

BigCuts는 사용하는 GPU 클러스터의 규모 및 종류와 상관없이 대규모의 딥 러닝 모델을 학습시키는 기법을 제시한다. 이를 위해 스토리지 시스템을 사용한 Tensor 관리 기법, 성능 모델에 기반한 구현 Design choice 선택 기법을 제시한다. 제시된 기법에 기반하여 주어진 GPU 클러스터의 구조와 학습하고자 하는 모델의

구조에 따라 최적화된 병렬화/실행 방법을 탐색할 수 있음을 보인다.

주요어: GPU, 딥 러닝, 코드 생성, 병렬화, 클러스터, 성능 모델링
학번: 2012-20860

Acknowledgements

관악에 첫발을 내디딘 때가 아직도 어제처럼 선연히 기억이 나는데, 어느새 13년의 시간이 지나 학업을 마무리하고 학교를 떠나게 되었습니다. 지난 시간을 되짚어 보니 행복하고 즐거운 순간들도 수없이 많이 떠오르지만, 방향을 잃고 힘겹게 헤맸던 순간들도 같이 떠오릅니다. 저 혼자의 능력과 의지만으로는 힘든 시간을 이겨내기 어려웠을 것 같습니다. 이 지면을 빌려 저에게 방향을 제시해 주시고 힘이 되어주신 모든 분께 감사의 말씀을 드리고자 합니다.

가장 먼저 지도교수님이신 이재진 교수님께 감사의 말씀을 드립니다. 교수님께서는 때로는 장난스럽고 때로는 엄하게 저를 지도해주셨고, 지식, 경험뿐 아니라 삶을 대하는 자세에 대해서도 많은 가르침을 주셨습니다. 특히 매사에 잔꾀를 부리지 않고 할 수 있는 한의 최선을 다해야 한다는 가르침, 남들과 비교하지 말고 자기 자신이 얼마나 더 나은 사람이 되었는지만을 확인하라는 가르침은 제 인생관의 많은 부분을 바꾸어 놓았습니다. 교수님의 지도가 없었다면 철없는 학부생이던 제가 한 사람의 공학 박사로 성장할 수 없었을 것입니다. 오랜 시간 저를 올바른 방향으로 이끌어 주셔서 다시 한번 진심으로 감사드립니다.

박사 학위 논문을 심사해주신 김진수 교수님, 문수묵 교수님, 정창희 교수님, 조형민 교수님께도 감사의 인사를 드립니다. 심사위원 교수님께서 보여주신

날카로운 통찰과 방향 제시 덕분에 제 박사 졸업 논문을 더 나은 방향으로 완성할 수 있었습니다. 또한 함께 공동 연구를 진행하며 미숙하던 저에게 많은 가르침을 주신 박종수 박사님께도 감사의 인사를 드립니다.

이재진 교수님 밑에서 동문수학하며 저와 추억을 함께 해 주신 연구실 동료분들께도 감사의 마음을 전합니다. 강수연, 권형달, 김선유, 김예하, 김정욱, 김정원 박사, 김정현 박사, 김진표, 김형모, 김홍규, 김홍준 박사, 김희훈, 나동진, 나정호, 도영동, 박대영, 박민혜, 박선명 박사, 박정호 박사, 박지영, 서보준, 서상민 박사, 손영준, 손장현, 신승훈, 신재호, 안규수, 오평석, 이용준, 이일구, 이준, 이지수, 임기현, 임현재, 정재훈, 조강원 박사, 주진영, 최영은, 한민희, 홍재기, Christophe Dubach 박사, Thanh Tuan Dao 박사까지, 연구와 휴식을 함께 하며 많은 영감을 주고받고 서로의 성장을 도왔습니다. 특히 연구실의 대소사를 해결할 때 저에게 많은 도움을 준 김희훈, 지루한 박사과정의 마지막 즈음에 즐거운 시간을 함께 한 김진표, 긴 시간 함께 연구하며 국내외에서 많은 추억을 쌓은 박정호 박사, 그리고 10년 넘게 중요한 일과 사소한 일을 가리지 않고 많은 이야기를 나누며 긴 시간을 함께한 친우 조강원 박사에게 좀 더 깊은 친애와 감사의 인사를 드립니다. 행정적인 일을 처리할 때 도움을 주신 손효정, 이은경, 이선주 선생님께도 감사의 인사를 드립니다.

자주 얼굴을 보지는 못했지만 사소한 이야기를 주고받으며 일상에서 소소한 웃음을 짓게 해준 친구들에게도 감사의 인사를 전합니다. 김용휘, 변정욱, 이재우, 임홍순, 조문장, 최준 덕분에 긴 시간 학위에 대한 압박감과 스트레스를 잊으며 즐거운 시간을 보낼 수 있었습니다. 앞으로의 삶에서도 즐거움을 나눌 수 있으면 좋겠습니다.

그리고 손주, 조카, 사촌의 성장을 지켜봐 주신 모든 친척분들께도 감사의 인사를 드립니다. 특히 박사 과정 중에 하늘나라로 가신 할아버지, 외할아버지께

부족한 손주가 감사 인사를 보내드립니다. 제가 학위를 받는 모습을 직접 보셨으면 많이 기뻐하셨을 것 같은데 그러지 못해 깊은 아쉬움이 있습니다. 하늘나라에서 기특한 마음으로 지켜봐 주실 것이라 믿습니다.

그리고 이 자리까지 오기까지 저를 물심 양면으로 지원해주시고 사랑으로 저를 키워주신 부모님께 가장 큰 감사의 마음을 전해드립니다. 자기 이야기를 잘 하지 않는 무심한 아들 때문에 기약 없는 박사과정 동안 많이 답답하고 궁금하셨을 텐데도, 말없이 믿고 바라봐주셔서 많이 죄송하고 감사합니다. 아직 많은 부분에서 부족한 저에게 끊임없이 세상을 보는 방법을 가르쳐 주시는 아버지, 최고의 지원과 사랑을 주셨음에도 항상 베풂이 부족했다고 아쉬워하시는 어머니께, 제가 이룬 모든 것은 두 분의 사랑 덕분에 있을 수 있는 것이었다고 말씀드리고 싶습니다. 다시 한번 감사드리고, 사랑합니다.

마지막으로, 지난 12년간 제 삶의 모든 순간을 공유하며 고난과 행복을 함께한 전보영에게 항상 고맙고 사랑한다고 말하고 싶습니다. 앞으로의 인생에서도 지금까지와 같이 서로가 서로에게 가장 큰 행복과 즐거움이 되었으면 좋겠습니다.

성인이 된 이후로 떠나본 적이 없는 관악산을 벗어나, 설렘과 두려움을 안고 새 발길을 내딛고자 합니다. 다소 길었던 학교에서의 삶과 경험이 앞으로의 인생에서의 보람과 성취를 위한 이정표가 될 것을 믿습니다. 지금까지 도움을 준 모든 분께 다시 한번 감사의 인사를 드리며, 받은 은혜에 보답할 수 있는 삶을 살아가도록 하겠습니다.

2021년 8월, 정우근