DeepCuts: A Deep Learning Optimization Framework for Versatile GPU Workloads

Wookeun Jung
Department of Computer Science and Engineering
Seoul National University, Korea
wookeun@aces.snu.ac.kr

Thanh Tuan Dao
Department of Computer Science and Engineering
Seoul National University, Korea
thanhtuan@aces.snu.ac.kr

Jaejin Lee
Department of Data Science
Department of Computer Science and Engineering
Seoul National University, Korea
jaejin@snu.ac.kr

Abstract

Widely used Deep Learning (DL) frameworks, such as TensorFlow, PyTorch, and MXNet, heavily rely on the NVIDIA cuDNN for performance. However, 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. In this paper, we propose a DL optimization framework for versatile GPU workloads, called DeepCuts. It considers both 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 considering kernel implementation parameters and GPU architecture parameters. The evaluation result with various DL workloads for inference and training indicates that DeepCuts outperforms cuDNN/cuBLAS-based implementations and the state-of-the-art DL optimization frameworks, such as TVM, TensorFlow XLA, and TensorRT.

CCS Concepts: • **Software and its engineering → Source code generation**; *Compilers*; • **Mathematics of computing** → *Mathematical software performance.*

Keywords: GPU, Deep Learning, Code Generation

ACM Reference Format:
Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. 2021. DeepCuts: A Deep Learning Optimization Framework for Versatile GPU Workloads. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454038

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454038

1 Introduction

GPUs are the *de facto* standard to run Deep Learning (DL) applications. Almost every widely used DL frameworks, such as TensorFlow[9], PyTorch[35], and MXNet[13], support GPU acceleration via cuDNN[17]. It is the state-of-the-art DL primitive library to accelerate DL computations.

However, cuDNN-based DL frameworks do not always give the best performance because of the following two reasons. One is the versatility of DL workloads. While 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 them. The input batch size also matters. Since the batch size is determined by the environment where the DNN model is used, it is hard to handle all batch sizes equally well only with cuDNN.

The other is that cuDNN has limited kernel fusion functionality. Kernel fusion is a well-known optimization technique that reduces 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 not sufficient to handle 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 [14, 19, 28, 32, 42]. 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[28], TensorRT[32]) achieve competitive or better performance than cuDNN for some 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 an 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 big to handle.

To this end, we propose DeepCuts, a DL optimization framework 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. One is that DeepCuts directly uses the information of the underlying architecture (*e.g.*, the size of GPU shared memory) to build a performance estimator. The other is that, 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 very hard 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 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. Then it executes the generated kernels and selects the best performing kernel for each DL operation or fused DL operation in the DL workload.

Major contributions of this paper are as follows:

- DeepCuts provides 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 high performance and versatility in the DL workloads.
- We propose a novel performance-model-driven code generation algorithm that considers two critical factors for performance improvement, fusion and parameter search, simultaneously. The algorithm searches for the best-performing kernel fusion method by considering the best-performing implementation parameters in a relatively small amount of time thanks 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 only with 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.

2 Related Work

Widely-used conventional DL frameworks, such as PyTorch [35], TensorFlow[9], and MXNet[13] 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[28] from Google, TensorRT[32] from NVIDIA, TVM[14], and Tensor Comprehensions[42]. 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[28] and TensorRT[32] 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[32] 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[42] is a domain-specific language that specifies DL operations. It automatically tunes

Table 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.

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 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[19] 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[26] 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[12, 36, 41, 43, 44] propose techniques to find optimized code for some important DL operations (*e.g.*, matrix multiplication and convolution). Widely-used BLAS

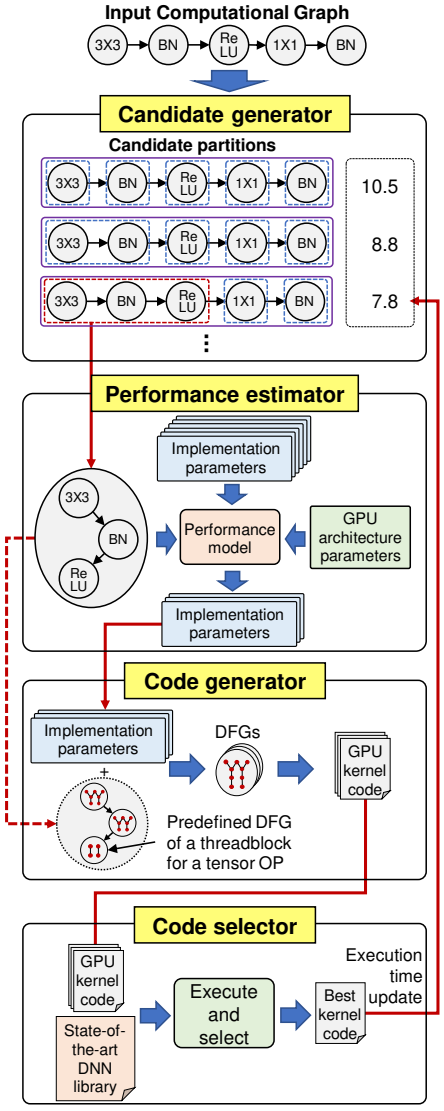


Figure 1. Overall workflow of DeepCuts. DeepCuts takes the whole computational graph of given workload and generates corresponding GPU kernels.

libraries[43, 44] 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[12] and KernelTuner[41] use evolutionary algorithms to find the optimal parameters. Pfaffe *et al.*[36] combines evolutionary algorithms and polyhedral models to generate the optimal kernel code.

Previous kernel fusion studies inspired and affected DeepCuts. Alwani *et al.*[10] 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.*[27] 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.

3 Overall Structure of DeepCuts

Figure 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[17]). The input graph is similar to the computational graph of PyTorch[35] or TensorFlow[9].

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.

```
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 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[17], 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 instance, the computational graph of the inference workload of VGG-16[39] 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 $\{\{\{A\}, \{B\}, \{C\}\}, \{\{A, B\}, \{C\}\}, \{\{A\}, \{B, C\}\}, \{\{A, B, C\}\}\}$. After 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 4. The detailed code generation algorithm is explained in Section 5 and Section 6.

Table 2. 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

Table 3. 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 4. 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

Target tensor operations. Table 2 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.

4 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[45] 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.

4.1 Kernel Implementation Parameters

The kernel implementation parameters considered in DeepCuts are described in Table 3 and Table 4. The parameters listed in Table 3 (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 4 are variable parameters. DeepCuts tries to find a well-performing combination of variable parameters during the optimization. The first and second rows in Table 4 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.

4.2 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[45]. For a given tensor operation, the operational intensity of its GPU kernel varies a lot depending on the implementation parameters.

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 6. It also computes the number of global memory transactions in a thread block using the formulae listed in Table 8. The formulae listed in Table 6 and Table 8 uses kernel implementation parameters listed in Table 4 and the target GPU architecture parameters listed Table 5. 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 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

Table 5. 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 6. 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 7. 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.	$2 \cdot N_{thread} \cdot K_{thread} \cdot H_{thread} \cdot W_{thread}$

Table 8. 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.	$2 \cdot N_{block} \cdot K_{block} \cdot H_{block} \cdot \lceil W_{block}/TRANS_{global} \rceil$

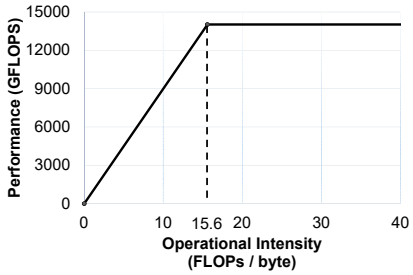


Figure 2. Roofline model of Nvidia Tesla V100.

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 2 shows the roofline model[45] of NVIDIA Tesla V100. It indicates that, when the operational intensity is less than 15.6, the 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[33]. 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 6, 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 7

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:

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

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

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 4. $COEF_r$ is computed as follows:

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

4.3 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 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[23], matrix multiplication in BERT[21]) 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.

4.4 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.

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.

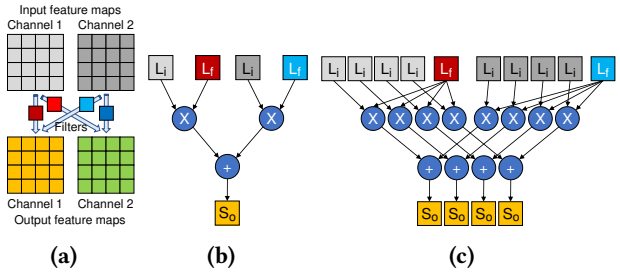


Figure 3. Baseline DFG construction. (a) 1×1 convolution operation with 2 input channels and 2 output channels. (b) The DFG of an output pixel when $C_{input} = 2$. (c) The baseline DFG for a thread block when $C_{input} = 2$.

5 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*.

5.1 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.

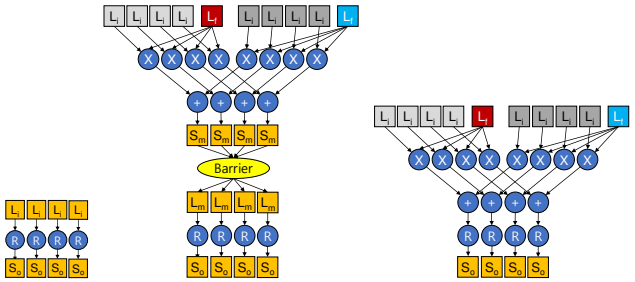


Figure 4. DFG concatenation. (a) Baseline DFG for ReLU operation. (b) Shared-memory-level fusion of a 1×1 convolution and a ReLU operation. (c) Register-level fusion of a 1×1 convolution and a ReLU operation.

For example, suppose the set S of implementation parameters selected by the performance estimator is given by,

$$S = \{N = 1, C = 2, K = 2, W = 4, H = 4, F_W = 1, F_H = 1,$$
$$P_W = 0, P_H = 0, S_W = 1, S_H = 1,$$
$$N_{block} = 1, K_{block} = 1, W_{block} = 2, H_{block} = 2, C_{input} = 2,$$
$$N_{thread} = 1, K_{thread} = 1, W_{thread} = 2, H_{thread} = 1\}.$$

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(a)). Then the computation for an output pixel of this tensor operation is represented by a DFG shown in Figure 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(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(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 4 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).

5.2 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 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 4.4, at the shared memory level and at the register level) determined by the performance model. Figure 4(b) shows the result of of concatenating the two DFGs in Figure 3(c) and Figure 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 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.

5.3 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 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.

6 GPU Kernel Code Generation

6.1 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[16]. 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 5), 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.

6.2 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.

Table 9. 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

Table 10. Deep Learning Benchmark Applications

	Architecture	Dataset
CNN	ResNet-50[24]	ImageNet Dataset[20]
	DenseNet-121[25]	
	Inception-v3[40]	
	MobileNetV2[38]	
RNN	DeepSpeech2[11]	AN4[4]
	Attention-RNN[30]	WMT'17[6]
MLP	Facebook-DLRM[31]	Kaggle Display Advertising Challenge Dataset [5]
	BERT[21]	SQUAD 1.1[37]

7 Evaluation

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.

7.1 Evaluation Environment

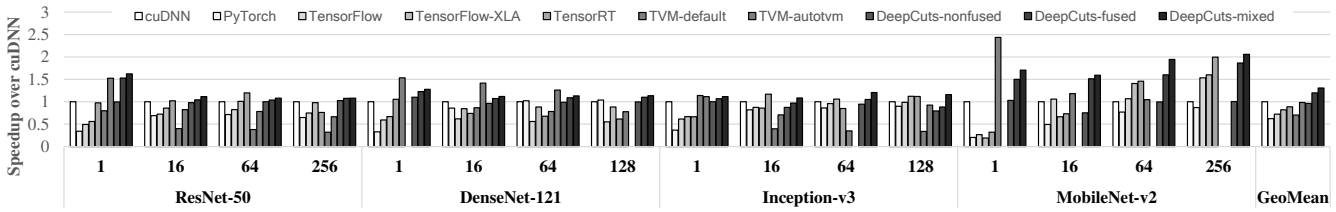
To check if DeepCuts and its performance model generically work for different GPU architectures, we evaluate DeepCuts on two different GPU architectures: NVIDIA Volta (V100) and Turing (RTX 2080). The system configuration and software tools used in the evaluation and their versionsis summarized in Table 9).

Benchmark applications. Benchmark applications for the evaluation are summarized in Table 10. We denote transformer-based MLPs as MLPs hereafter.

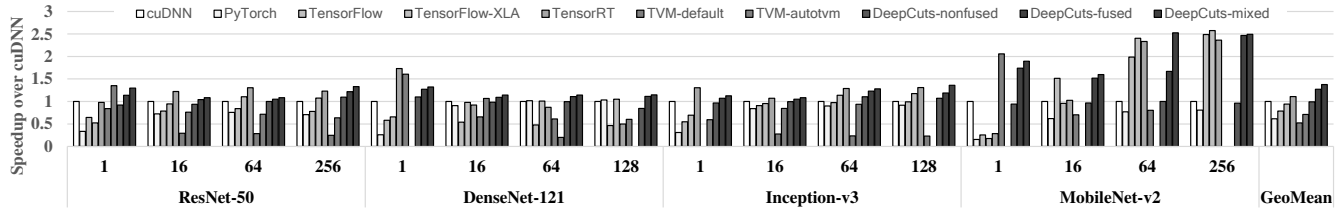
Measurements. We measure the performance of both training and inference workloads for each benchmark. For non-CNN benchmarks, we measure the execution time for one training epoch. For CNNs whose execution is very regular, we measure the training time for 100 mini-batches. We exclude the overheads for 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 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.

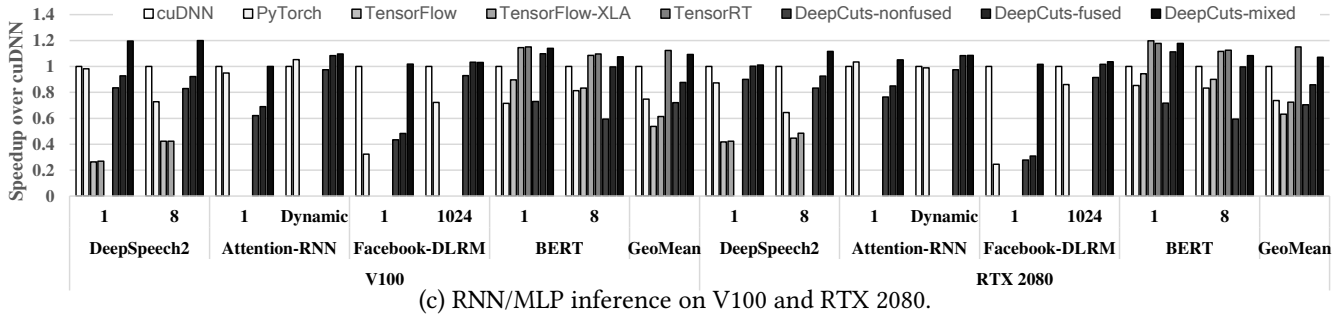
DeepCuts versions. We evaluate three different versions of DeepCuts: DeepCuts-no-fusion, DeepCuts-fusion, and



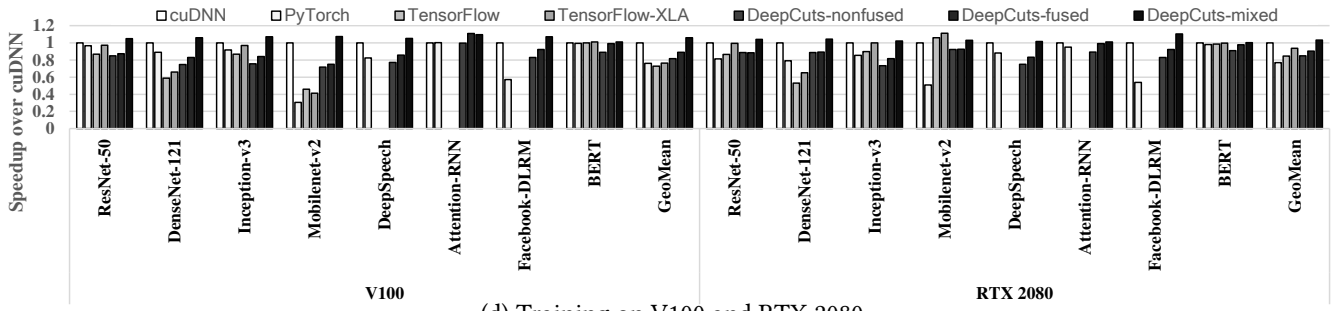
(a) CNN inference on V100.



(b) CNN inference on RTX 2080.



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



(d) Training on V100 and RTX 2080.

Figure 5. Speedup over cuDNN for inference and training.

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[47] .

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[7, 8] for CNNs and DeepSpeech2. In the case of DeepSpeech2, we use TensorFlow 1.14 because the official tutorial code for DeepSpeech2 is not working with TensorFlow 2.4.1. We use the reference code provided by NVIDIA[3] for BERT.

We use TensorRT-integrated TensorFlow[1] 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[2].

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[18]. For non-CNN models, we use scripts provided by the author of the original paper[22] or from some widely-used projects[34].

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.

7.2 Evaluation Results

Overall performance. Table 11 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.

Table 11. # 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 12. # 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

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 frameworks 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(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

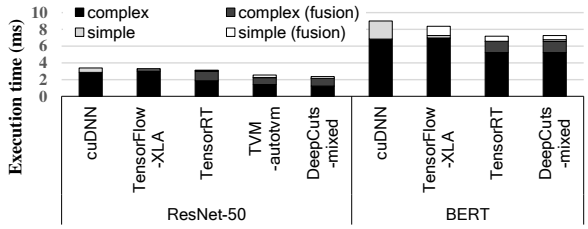


Figure 6. Exec. time breakdown for ResNet-50 and BERT.

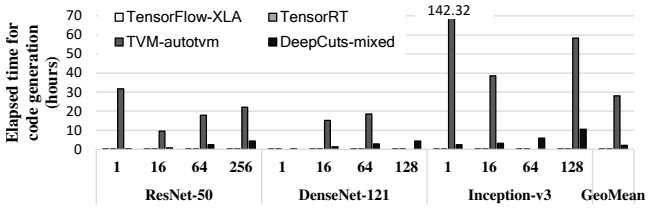


Figure 7. Elapsed time for code generation. The bars for the elapsed times of TensorFlow XLA and TensorRT are almost invisible because they take only a few seconds.

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(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.

RNN/MLP inference. Figure 5(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(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.

7.3 Execution Time Breakdown

Figure 6 shows the execution time breakdown of GPU kernels for ResNet-50 and BERT inference measured with the NVIDIA nvprof profiler[33]. 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 2. 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 6.

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.

7.4 Code Generation Time

Figure 7 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.

8 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[46] and Octave convolution[15]. 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 performing 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[29]. 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.

9 Conclusions

In this paper, we introduce DeepCuts, a DL optimization framework for versatile GPU workloads. 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. We are going to open the source code of DeepCuts to the public.

Acknowledgement

This work was supported in part by the National Research Foundation of Korea (NRF) grants (No. NRF-2016M3C4A7952587 and No. NRF-2019M3E4A1080386) and by the Institute for Information & communications Technology Promotion (IITP) grant (No. 2018-0-00581, CUDA Programming Environment for FPGA Clusters), all funded by the Ministry of Science and ICT (MSIT) of Korea. ICT at Seoul National University provided research facilities for this study.

References

[1] Accelerating Inference In TF-TRT User Guide. https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html. Accessed: 2020-04-14.

[2] BERT Example using the TensorRT C++ API. https://github.com/NVIDIA/TensorRT/tree/release/5.1/demo/BERT/. Accessed: 2020-04-14.

[3] BERT For TensorFlow. https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/LanguageModeling/BERT. Accessed: 2020-04-17.

[4] The CMU Audio Databases. http://www.speech.cs.cmu.edu/databases/an4/index.html. Accessed: 2019-4-15.

[5] Display Advertising Challenge. https://www.kaggle.com/c/criteo-display-ad-challenge/overview. Accessed: 2019-4-15.

[6] EMNLP 2017 SECOND CONFERENCE ON MACHINE TRANSLATION (WMT17). http://www.statmt.org/wmt17/translation-task.html. Accessed: 2019-4-15.

[7] TensorFlow implementation of DeepSpeech2. https://github.com/yao-matrix/deepSpeech2. Accessed: 2019-11-21.

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

[9] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://doi.org/10.5555/3026877.3026899

[10] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 22. https://doi.org/10.1109/MICRO.2016.7783725

[11] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. 2016. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*. 173–182. https://doi.org/10.5555/3045390.3045410

[12] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 303–316. https://doi.org/10.1145/2628071.2628092

[13] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594. https://doi.org/10.5555/3291168.3291211

[15] Yunpeng Chen, Haoqi Fan, Bing Xu, Zhicheng Yan, Yannis Kalantidis, Marcus Rohrbach, Yan Shuicheng, and Jiashi Feng. 2019. Drop an Octave: Reducing Spatial Redundancy in Convolutional Neural Networks With Octave Convolution. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 3434–3443. https://doi.org/10.1109/ICCV.2019.00353

[16] John Cheng, Max Grossman, and Ty McKercher. 2014. *Professional Cuda C Programming*. John Wiley & Sons. https://doi.org/10.5555/2935593

[17] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[18] Soumith Chintala. ImageNet training in PyTorch. https://github.com/pytorch/examples/tree/master/imagenet. Accessed: 2019-11-21.

[19] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar Vijay, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. *CoRR* abs/1801.08058 (2018). arXiv:1801.08058 http://arxiv.org/abs/1801.08058

[20] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

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

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[25] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708. https://doi.org/10.1109/CVPR.2017.243

[26] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62. https://doi.org/10.1145/3341301.3359630

[27] Wonkyung Jung, Daejin Jung, Sunjung Lee, Wonjong Rhee, Jung Ho Ahn, et al. 2018. Restructuring batch normalization to accelerate CNN training. *arXiv preprint arXiv:1807.01702* (2018).

[28] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. https://www.youtube.com/watch?v=kAOanJczHA0. *TensorFlow Dev Summit* (2017). Accessed: 2021-05-10.

[29] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2020. Automatic Horizontal Fusion for GPU Kernels. *CoRR* abs/2007.01277 (2020). arXiv:2007.01277 https://arxiv.org/abs/2007.01277

[30] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. *CoRR* abs/1508.04025 (2015). arXiv:1508.04025 http://arxiv.org/abs/1508.04025

[31] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). arXiv:1906.00091 http://arxiv.org/abs/1906.00091

[32] NVIDIA. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt. Accessed: 2020-05-11.

[33] NVIDIA. NVIDIA visual profiler. https://developer.nvidia.com/nvidia-visual-profiler.

[34] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*. https://doi.org/10.18653/v1/N19-4009

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *CoRR* abs/1912.01703 (2019). arXiv:1912.01703 http://arxiv.org/abs/1912.01703

[36] Philip Pfaffe, Tobias Grosser, and Martin Tillmann. 2019. Efficient hierarchical online-autotuning: a case study on polyhedral accelerator mapping. In *Proceedings of the ACM International Conference on Supercomputing*. ACM, 354–366. https://doi.org/10.1145/3330345.3330377

[37] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100, 000+ Questions for Machine Comprehension of Text. *CoRR* abs/1606.05250 (2016). arXiv:1606.05250 http://arxiv.org/abs/1606.05250

[38] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation. *CoRR* abs/1801.04381 (2018). arXiv:1801.04381 http://arxiv.org/abs/1801.04381

[39] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition, Vol. abs/1409.1556. arXiv:1409.1556 http://arxiv.org/abs/1409.1556

[40] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826. https://doi.org/10.1109/CVPR.2016.308

[41] Ben van Werkhoven. 2019. Kernel Tuner: A search-optimizing GPU code auto-tuner. *Future Generation Computer Systems* 90 (2019), 347–358. https://doi.org/10.1016/j.future.2018.08.004

[42] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[43] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188. https://doi.org/10.1007/978-3-319-06486-4_7

[44] R Clinton Whaley and Jack J Dongarra. 1998. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 38–38. https://doi.org/10.1109/SC.1998.10004

[45] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[46] Bichen Wu, Alvin Wan, Xiangyu Yue, Peter Jin, Sicheng Zhao, Noah Golmant, Amir Gholaminejad, Joseph Gonzalez, and Kurt Keutzer. 2018. Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 9127–9135. https://doi.org/10.1109/CVPR.2018.00951

[47] Lianmin Zheng and Eddie Yan. Auto-tuning a convolutional network for NVIDIA GPU. https://docs.tvm.ai/tutorials/autotvm/tune_relay_cuda.html. Accessed: 2019-11-21.