Accelerating the inference speed of a DNN model using CUDA

Shilong Dong, Zhenghan Nan, Yifu Deng

1 Abstract

When deploying deep learning algorithms for real-time applications, computation efficiency must be exceedingly high to meet the required latency constraints. Sentiment analysis, for example, poses significant challenges, particularly in text data processing, which demands both high accuracy and speed as a critical component of analytics. In this work, we demonstrate the optimization of a PyTorch-based sentiment analysis model by creating a more efficient version on a CUDA platform. Specifically, techniques such as shared memory utilization and stream optimization deliver remarkable performance gains. As shown in our experiments, these optimizations achieve approximately a 500-fold improvement in single-case inference performance compared to the original C++ implementation, while maintaining the model's inference accuracy at 84%. These results underscore the immense benefits of GPU acceleration and pave the way for further advancements in optimizing deep learning inference.

2 Introduction

The importance of deep learning for various applications, especially those requiring running models in a device, has recently increased considerably. This is because the efficiency of operation of such models is nearly single-handedly settled by the end user's experience and the functioning of the entire system. Much higher delays are associated with traditional re-implementation on CPUs and developing deep learning algorithms using high-level frameworks than if the models were to be implemented directly onto the devices. This is, however, more severe in places where the vast majority of businesses operate under ideal conditions with low turnaround expectations yet require a lot of operating power.

These issues are illustrated with sentiment analysis, a primary task in Natural Language Processing. The demand for precision and low-cost time is evident with the growing variety of applications, such as social media sentiment analysis and product comment systems, that require fast and precise sentiment analysis in real-time. Luckily, with better technology, it can be done, though issues abound, such as the high computational power that models of neural networks demand to ensure precision.

This document addresses the abovementioned issues by discussing a framework intended to reduce the time required for deep learning model inference by leveraging the CUDA parallel computing architecture. Our approach is to re-implement a given sentiment analysis model developed in PyTorch as an optimized CUDA version, especially the parts performing concurrent matrix multiplications, activation functions, and other

computations. By optimizing how threads access memory, adjusting thread block deployments, and employing kernel fusion, we significantly reduce the computational burden while not changing the accuracy of the model.

The experimental results underscore the validity of this approach with at least 500 times speed-up, achieving a total inference time of 0.1624ms on a single case as opposed to 110.785ms on the previous C++ implementation. Combining these optimizations increased shared memory usage by 9%, memory coalescing by 17%, and stream optimization providing marginal improvement, improving all of this while the model maintained an accuracy of 84% in prediction.

In the subsequent section, we discuss prior work done on enhancing sentiment analysis models, pushing the models onto the hardware, and optimizing using CUDA. Section IV then reiterates our approaches, outlining the step-by-step strategies we are employing to solve the case of deep learning on CUDA more efficiently.

Section V presents our setup of the experiments by describing the hardware configurations, the development environment, and a detailed description of the evaluation metrics. Section VI describes our experiments and their analysis, where the performance across different implementations is compared and the effectiveness of multiple optimizing techniques is evaluated.

To conclude, Section VII summarizes the main results of our experiments and provides some recommendations for further studies in the area of GPU acceleration for deep learning inferencing.

3 Literature Survey

Transitioning deep learning models from academic research to practice presents serious hurdles, particularly in environments where low inference latency and minimal use of system resources are essential. This segment reviews existing acceleration techniques of deep learning models in the context of sentiment analysis, on-device model deployment approaches, and CUDA optimization techniques.

3.1 Sentiment Analysis Acceleration

One of the illustrative case studies demonstrating such problems is sentiment analysis, regarded as one of the elementary tasks in NLP requiring the analysis of text streams in real-time Dang et al. [2020]. Since such models usually consist of several matrix multiplications through a feed-forward neural network, they lend themselves to parallel processing optimization. The fields of application are vast, starting with social network tracking and ending with the analysis of customer reviews, where high accuracy and fast processing time are both critical.

3.2 On-device Model Deployment

The studies involving data local execution, mainly focused on devices such as NVIDIA Jetson Orin, have indicated various ways to achieve effective model inferencing. Qiu et al. [2016] accelerated the process using FPGAs, but it meant they had to hire people knowledgeable in such hardware. Chen et al. [2016] put forward two designs for a recent Eyeriss, a custom neural processing unit, and even demonstrated one of its potential custom hardware advantages. The research by Jouppi et al. [2017] on Google's TPU was

a practical demonstration of the effectiveness of special-purpose hardware. Still, most of these implementations are mainly aimed at optimizing a specific platform without taking into account the parallel processing resources of GPUs to their full extent.

3.3 CUDA Optimization Methods

The GPU parallelization and implementation come with excellent performance enhancement due to hardware-level parallelism optimization. Volkov and Demmel [2008] formed basic principles that aid polynomials of matrices, which provide crucial components for successful deployment of deep learning. Other authors Lai and Seznec [2013] showed how to achieve maximum efficiency of memory bandwidth utilization through appropriate memory access patterns. Sun et al. [2024] demonstrated the application of kernel fusion techniques whereby memory bandwidth usage was diminished as performance was appealingly enhanced. These investigations form the requisite core building blocks critical for CUDA acceleration on GPUs. However, they focus on single processes more than the complete workflow optimization.

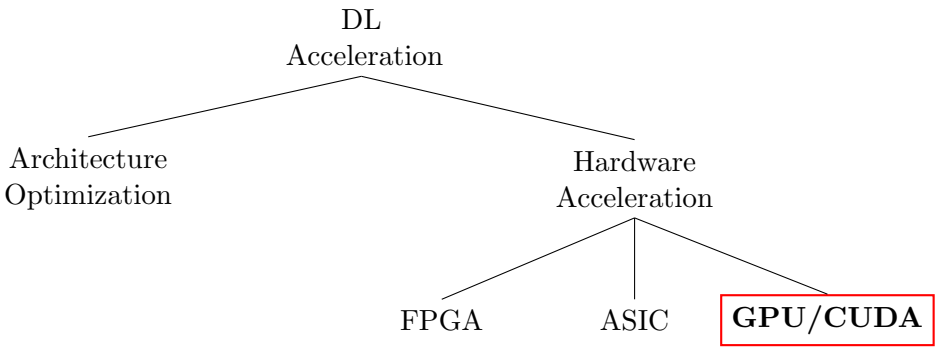

Figure 1: Taxonomy of Deep Learning Acceleration Approaches

In the taxonomy displayed in Figure 1, an emphasis on GPU/CUDA optimization, depicted in red, is the theme of this work developing upon other highlighted material. In this context, we focus our attention on improvements aiming at increasing the speed of Graphical Processor Unit (GPU)-based deep learning inference acceleration by outlining a systematic approach to optimizing CUDA routines. The subsequent section introduces this systematic approach, noting that each part of the system has been designed to target a particular performance bottleneck while maintaining the accuracy of the model.

4 Proposed Idea

Our approach focuses on optimizing the inference process of a sentiment analysis neural network through CUDA acceleration. The critical insight is systematically identifying and parallelizing computational bottlenecks while maintaining the model's accuracy. We propose a three-stage optimization workflow as shown in Figure 2:

First, we develop and deploy a neural network-based model targeted at sentiment analysis. Given the attention to maximizing expressiveness while reducing complexity and making the model hardware agnostic, the model's architecture consists of stacks of linear layers followed by the ReLU activation function.

Second, we specify a PyTorch model and then code it in C++ using the Eigen library for matrix calculations. We provide a loading method for the binary weights to ensure that the parameters trained in PyTorch are loaded correctly into C++-applied methods.

Finally, we identify computationally intensive operations—particularly matrix multiplications and element-wise operations—that can benefit from parallel execution on GPU hardware. This lays the groundwork for CUDA acceleration while maintaining architectural consistency across implementations.

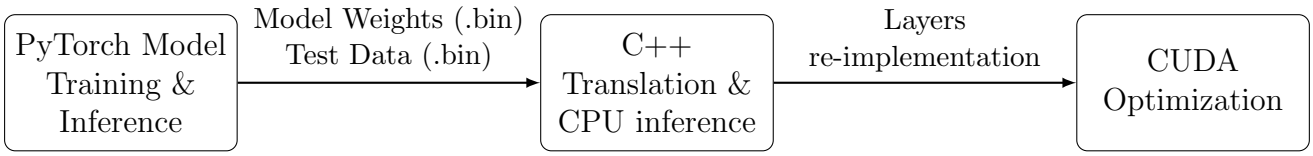

Figure 2: Optimization Workflow Overview

4.1 Model Design & Training

On the one hand, our sentiment analysis model ensures accuracy, while on the other hand, it is computationally efficient. The network architecture is optimized to extract relevant features and yet allows for an increase in parallelism during inference. The model takes text as input, and it is translated through a series of fully connected layers and activation functions, which continuously lower the dimension of the vector while retaining its meaning.

4.1.1 Network Architecture

The model consists of two main components:

- Feature Extractor:
  - Input Layer: Accepts 256 tokens $\times$ 300-dimensional GloVe embeddings
  - First Linear Layer: 300 $\rightarrow$ 150 dimensions with ReLU activation
  - Second Linear Layer: 150 $\rightarrow$ 64 dimensions with ReLU activation
  - Mean Pooling: Aggregates token-level features into sentence representation
- Classifier:
  - First Linear Layer: 64 $\rightarrow$ 32 dimensions with ReLU activation
  - Output Layer: 32 $\rightarrow$ 2 dimensions for binary sentiment classification

The specified structure for progressive dimension reduction (300$\rightarrow$150$\rightarrow$64$\rightarrow$32$\rightarrow$2) serves several goals. First, it provides a framework for effective feature compression by successfully reducing the dimensionality of the input space. Second, the mean-pooling operation considerably decreases the demand for memory space and the computational burden for the classifier stages. Lastly, this makes the linear layers followed by ReLU activations most suitable for GPU acceleration since these layers can be efficiently performed in parallel.

4.1.2 Training Implementation

The IMDB movie reviews dataset with 25,000 reviews was used as a sample to demonstrate how the PyTorch framework greatly improved the model training process within the proposed training pipeline. The models were trained using 5 epochs and a batch size of 8, and the cross-entropy loss function with an Adam optimizer set to a learning rate of 0.001. We tokenize the text data at a fixed length of 256, where each token is represented by 300-dimensional GloVe embeddings.

4.2 C++ Translation & Inference

4.2.1 Translation Implementation

The translation from PyTorch into C++ seeks to not only provide a numerical invariant but also an efficient computation for the end user. We develop a binary weight export where both parameter values and the shape information are encoded. The matrix functions employ the Eigen library, which is known for its efficient implementations of linear algebra routines and readiness for SIMD parallelization. Row-major representation of the matrix is kept in order to implement the most efficient memory access and use caches when it is practical.

4.2.2 Inference Implementation

The core inference algorithm is implemented as follows:

Algorithm 1 Sentiment Classification Inference

Require: Input text embeddings matrix $X \in \mathbb{R}^{256\times300}$
Ensure: Binary sentiment prediction
// **Feature Extraction**
$X_1 \leftarrow \text{ReLU}(XW_1 + b_1)$ ▷ $X_1 \in \mathbb{R}^{256\times150}$
$X_2 \leftarrow \text{ReLU}(X_1W_2 + b_2)$ ▷ $X_2 \in \mathbb{R}^{256\times64}$
// **Mean Pooling**
$f \leftarrow \text{MeanPool}(X_2)$ ▷ $f \in \mathbb{R}^{64}$
// **Classification**
$h \leftarrow \text{ReLU}(fW_3 + b_3)$ ▷ $h \in \mathbb{R}^{32}$
$y \leftarrow hW_4 + b_4$ ▷ $y \in \mathbb{R}^2$
return y

Multiple optimization techniques refine the inference pipeline:
Memory Management. As part of the design, memory alignment for SIMD operations is properly addressed as the integration uses prior allocated matrix buffers with minimal memory transfers.
Computation Pipeline. Matrix operations are vectorized using Eigen and its other internal optimizations, augmented with batch operation and slow, unreduced ReLU.
I/O Optimization. The architecture also works with memory-mapped weight and efficient batch processing to reduce the workload concerning data conversion during inference time.
In this manner, the proposed implementation is fully prepared for GPU computations without a loss of accuracy when working with hyperpolarization classification tasks.

4.3 CUDA Optimization

We progressively improved our suggested CUDA algorithm. First, we made the straightforward CUDA implementation, which guarantees accuracy and validates the usefulness of CUDA acceleration. Next, we applied the tiling method, which was presented during the lecture, to optimize the many matrix multiplications in the linear layers of our model. Then, we made further improvements in saving memory access time by putting intermediate results into shared memory. Finally, we applied stream technology to lessen latency caused by moving data from one place to another by loading various data inputs to the GPU at the same time.

4.3.1 Naive Implementation

In this set of work, and like most such implementations, we primarily performed the following functions in the order in which they have been introduced in this manuscript:
Model Initialization. At the beginning, we created an instance of the model class specified in our C++ code. Then, in accordance with the standards of the PyTorch structure, we loaded all the model weights onto the GPU. This allowed the model's weights to be available for further operations. A pseudo-code for this step is provided in Algorithm 2.

Algorithm 2 Model Initialization

procedure InitializeGPUMemory(*model_weights*)
  for each layer l in model **do**
    $weight\_size \leftarrow$ `sizeof`(float) $\times$ l.weight.size
    $bias\_size \leftarrow$ `sizeof`(float) $\times$ l.bias.size
    $d\_weight\_l \leftarrow$ `cudaMalloc`($weight\_size$)
    $d\_bias\_l \leftarrow$ `cudaMalloc`($bias\_size$)
    `cudaMemcpy`(d_weight_l, l.weight, $weight\_size$, H2D)
    `cudaMemcpy`(d_bias_l, l.bias, $bias\_size$, H2D)
  end for
  return device pointers
end procedure

Input Data Transfer. For the sake of efficiency and optimization for this step, we prepared the required size of input data at the beginning of each case evaluation and opted for the use of *memcpy* commands to perform the transfer of the data to the GPU.
Model Feed-Forward in CUDA. This section addresses changes to the architecture in respect of several layers during the model's forward pass and their implementation in CUDA. The model is made up of four linear layers, which include weight computation comprising matrix multiplications, four linear layers where the bias is computed and matrix additions are utilized, and three ReLU activation functions. Of the four linear layers, the second does not contain any bias and also computes a mean pooling layer that converts the output to a vector while the previous layers output matrices. Hence, the implementation of our linear layers requires two methods: matrix-matrix multiplication and matrix-vector multiplication. The decision was made to code weight calculations in two different CUDA kernels that implement linear layers, while bias and ReLU activation are calculated in a single CUDA kernel. These implementations are presented in Algorithms 3 and 4.

4.3.2 Shared Memory Usage

After conducting extensive experiments on the naive implementation, we realized that Algorithm 3 could become a bottleneck since it did not fully utilize the hardware resources. To address this, we optimized these procedures with shared memory. Due to space limitations, we only show the matrix-vector multiplication part in Algorithm 5.

4.3.3 Stream Usage

To increase throughput in terms of model inference, we used the CUDA stream approach, which allowed the asynchronous execution of tasks on the GPU. We created multiple streams, each with its own pre-allocated device memory buffers, to enable independent input processing. Each stream performed an asynchronous forward pass of the model with its buffers, while at the same time other streams were busy with their computations, effectively combining calculations with data transfers.

Algorithm 3 Linear Layer Weight Calculation

procedure MATRIX-MATRIX MULTIPLICATION(A, B, C, m, n, k)
    for each thread in grid **do**
        $row \leftarrow$ `blockIdx.y` $\times$ `blockDim.y` $+$ `threadIdx.y`
        $col \leftarrow$ `blockIdx.x` $\times$ `blockDim.x` $+$ `threadIdx.x`
        if $row < m$ **and** $col < n$ **then**
            $value \leftarrow 0.0$
            for $e \leftarrow 0$ **to** $k - 1$ **do**
                $value \leftarrow value + A[row \times k + e] \times B[e \times n + col]$
            end for
            $C[row \times n + col] \leftarrow value$
        end if
    end for
end procedure

procedure MATRIX-VECTOR MULTIPLICATION(A, x, y, m, n)
    for each thread in grid **do**
        $row \leftarrow$ `blockIdx.x` $\times$ `blockDim.x` $+$ `threadIdx.x`
        if $row < m$ **then**
            $value \leftarrow 0.0$
            for $e \leftarrow 0$ **to** $n - 1$ **do**
                $value \leftarrow value + A[row \times n + e] \times x[e]$
            end for
            $y[row] \leftarrow value$
        end if
    end for
end procedure

Algorithm 4 Linear Layer Bias-n-ReLU Calculation

procedure ADD BIAS & RELU MATRIX($x, bias, m, n$)
  for each thread in grid **do**
    $idx \leftarrow$ `blockIdx.x` $\times$ `blockDim.x` + `threadIdx.x`
    $total \leftarrow m \times n$
    if $idx < total$ **then**
      $col \leftarrow idx \bmod n$
      $x[idx] \leftarrow x[idx] + bias[col]$
      if $x[idx] < 0$ **then**
        $x[idx] \leftarrow 0$
      end if
    end if
  end for
end procedure

procedure ADD BIAS & RELU VECTOR($x, bias, n$)
  for each thread in grid **do**
    $idx \leftarrow$ `blockIdx.x` $\times$ `blockDim.x` + `threadIdx.x`
    if $idx < n$ **then**
      $x[idx] \leftarrow x[idx] + bias[idx]$
      if $x[idx] < 0$ **then**
        $x[idx] \leftarrow 0$
      end if
    end if
  end for
end procedure

Algorithm 5 Linear Layer Weight Calculation Using Shared Memory

procedure M-V MULTIPLICATION SHARED MEMORY(A, x, y, m, n)
  Shared Memory Allocation: `shared_x`[64]
  $row \leftarrow$ `blockIdx.x` $\times$ `blockDim.x` + `threadIdx.x`
  if `threadIdx.x` $< n$ **then**
    `shared_x`[`threadIdx.x`] $\leftarrow x$[`threadIdx.x`]
  end if
  `__syncthreads()`
  if $row < m$ **then**
    $value \leftarrow 0.0$
    for $e \leftarrow 0$ **to** $n - 1$ **do**
      $value \leftarrow value + A[row \times n + e] \times$ `shared_x`$[e]$
    end for
    $y[row] \leftarrow value$
  end if
end procedure

5 Experimental Setup

5.1 Experiment Environment

5.1.1 Model Training and Weight Saving

We conducted the PyTorch model training and weight extraction on a MacBook Pro with Apple M1 Max CPU and 32GB unified memory, running macOS Monterey 12.6. The software environment included Python 3.9.16, PyTorch 2.0.1, along with `torchtext` and `NLTK` libraries. The model was trained on a subset of the IMDB sentiment analysis dataset (1000 training samples, 200 test samples) using the Adam optimizer with a learning rate of 0.001, a batch size of 8, and trained for 5 epochs with a dropout rate of 0.2.

5.1.2 Model Inference

Performance testing was conducted on the CIMS CUDA5 machine, which is equipped with an Intel(R) Xeon(R) E5-2650 v2 core CPU and an NVIDIA GeForce RTX 4070 GPU. The software stack included GCC 9.4.0 and CUDA 12.4.

5.2 Evaluation Metrics

Total Running Time. We use this metric to evaluate the total running time of a method mentioned previously on the whole test set, which includes the model initialization, model inference, and memory management time.
Single Case Inference Time. We use this metric to measure the model's inference time on a single data case input.
Classification Accuracy on Test Set. We use this metric to evaluate our adapted model's performance on the original sentiment analysis task.

6 Results and Analysis

6.1 Speedup on different iterations

Implementation	Time per Iteration (s)	Speedup
CPU (Baseline)	246.890	–
CUDA (Naive)	1.549	159
CUDA (Shared Memory)	1.406	175
CUDA (Stream Optimized)	1.401	176

Table 1: Performance Comparison of CPU and CUDA Implementations on cuda5

Using 20,000 test points, we calculated the time required to process each iteration. The results, which are included in Table 1, indicate that inference time has indeed been reduced through the use of the CUDA language tools. With the use of a rather basic CUDA model, an approximate acceleration of 159× was recorded, with the number of seconds per iteration being decreased from 246.890 to 1.549. The shared memory optimization provided approximately an additional 9 percent improvement in performance,

while the stream optimization gave an insignificant extra advantage. These results imply that GPU parallelism and appropriate memory access patterns are quite useful for enhancing the performance of large-scale inference.

6.2 Inference Speed of single case

Platform	Implementation	Time per Case (ms)
Mac M1 Max Chip	PyTorch (CPU)	0.0027
Mac M1 Max Chip	PyTorch (MPS)	0.0080
Intel Xeon E5-2650 v2	C++ (Baseline)	110.7850
Intel Xeon E5-2650 v2	CUDA (Naive)	0.2215
Intel Xeon E5-2650 v2	CUDA (Shared Memory)	0.1835
Intel Xeon E5-2650 v2	CUDA (Stream Optimized)	0.1624

Table 2: Inference Speed of Single Case on Different Platforms and Implementations (Time in ms)

We also determined the inference speed per single case for every implementation. The results in Table 2 depict the one-case inference time across various platforms and implementations. When the C++ baseline is deployed together with the CUDA implementations on the Intel Xeon E5-2650 v2 platform, it is observed that the CUDA implementations perform remarkably better. The baseline, when combined with CUDA, achieves approximately a 500-fold improvement with the naive CUDA version, and an additional 17 percent improvement through shared memory utilization.

However, the CUDA implementations are still slower than the implementation done with the PyTorch framework on the CPU of the Mac M1 Max chip. This implies that certain CUDA optimizations may benefit specific types of hardware, while highly optimized frameworks such as PyTorch with purpose-built hardware integration may be more beneficial for single-case inference. These results broadly suggest that it is crucial to optimize for the theoretical distribution, as small-batch operations may be limited by memory transfer latency and setup overhead.

6.3 Classification Accuracy

Due to our correct implementation and the preservation of floating-point precision, the model's classification accuracy showed no significant variation. Both the C++ implementation and various CUDA-based implementations achieved an accuracy of 84%.

7 Conclusions

7.1 Contributions

We discuss the use of PyTorch, the C++ Eigen library, and CUDA in model inference in the present study. We conducted extensive experiments on the CIMS server and performed model inference based on CUDA without any performance loss. Our CUDA code optimization process for the parallel programming model of computing was carried out in three stages.

First, a straightforward approach focused on correctness was implemented, where each layer previously defined in C++ was dealt with using basic methods. Next, several optimization strategies were applied, including tiling approaches to matrix multiplication algorithms and the use of shared memory to eliminate inefficient memory references. Finally, stream processing techniques were employed to reduce the time required for the inference engine to process input data.

With our naive implementation utilizing CUDA, a speedup of 150x on the entire test set and 500x for single-case inference indicates that the C++ Eigen-based implementation does not fully leverage the benefits of parallel computing achievable through CUDA. These results suggest that using CUDA for inference tasks is a highly effective decision.

Regarding GPU performance, shared memory optimization brought an improvement of approximately 20%, highlighting the correct formulation of the problem and the effective use of memory in the designed CUDA kernels as key factors enabling better performance. Furthermore, when combined with streams, CUDA shared memory offered a slight additional performance increase. However, this was expected, as the input test data was relatively small. Overlapping computation with asynchronous memory reading and writing did not provide significant time savings compared to the streaming overhead. That said, for larger and more complex input data sets, or models with more processing elements, the use of streams would likely yield more substantial benefits.

7.2 Limitations and Future Work

So far, our experiments have been restricted to a simplistic model, a small dataset, and tests based on linear layers with ReLU activation. We have not performed any tests more broadly on models within the mainstream Transformer architecture. Nevertheless, in the case of Transformer models, which heavily rely on multiplication, we expect our technique to yield good outcomes. Furthermore, a possible avenue for future research is to explore increasing the input data dimensions to multi-modal spaces, such as images or videos, which involve massive data. In these scenarios, CUDA's acceleration is likely to become even more critical.

One further limitation of our research is that, while our implementation clearly outperforms the C++ Eigen-based implementation by a significant margin, it is still approximately 20x-100x slower than frameworks such as PyTorch. This highlights the gap and the need to extend the investigation with additional techniques aimed at narrowing the performance differences between our implementation and the metrics achieved by widely used frameworks.

References

Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2016.

Nhan Cach Dang, María N Moreno-García, and Fernando De la Prieta. Sentiment analysis based on deep learning: A comparative study. *Electronics*, 9(3):483, 2020.

Norman P Jouppi, Clifford Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-

datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM, 2017.

Jun Lai and André Seznec. Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 1–10. IEEE, 2013.

Jing Qiu, Jian Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Yao Yu, Tiancheng Tang, Ning Xu, Sen Song, and Huazhong Yang. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35. ACM, 2016.

Wei Sun, Ang Li, Sander Stuijk, and Henk Corporaal. How much can we gain from tensor kernel fusion on gpus? *IEEE Access*, 2024.

Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–11. IEEE, 2008.