CNN performance analysis with different GPU libraries and Attention optimization on GPU using Tensor Core WMMA API

by

Zhumakhan Nazir

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science
at the
NAZARBAYEV UNIVERSITY
June 2024

Author . . . .
Department of Computer Science
April 24

Certified by . . . .
Jurn Gyu Park
Professor
Thesis Supervisor

Certified by . . . .
Minho Lee
Professor
Co-supervisor

Certified by . . . .
Zhanat Kappassov
Professor
External Examiner

Accepted by . . . .
Yelyzaveta Arkhangelsky
Dean, School of Engineering and Digital Sciences

CNN performance analysis with different GPU libraries and Attention optimization on GPU using Tensor Core WMMA API

by

Zhumakhan Nazir

Submitted to the Department of Computer Science
on April 24, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

Deep Learning has been very effective in tasks related to texts, images or time series data. With increased efficiency, demand for hardware and software capabilities has increased as well. Nvidia GPUs are the main hardware type that is used to both train and serve DL models of various sizes. It comes with high performance linear algebra (cuBLAS), deep neural networks (cuDNN) libraries and inference engines (TensorRT) which are used to accelerate computations. In addition to these, CUDA parallel programming software allows users to devise their own custom kernels for specific cases. This work consists of two parts. In the first part, three different implementations of Yolo, a convolutional neural network model, using cuBLAS, cuDNN and TensorRT were evaluated. By collecting the GPU performance metrics such as compute utilization and memory throughput, the most important metrics that greatly affect the performance of kernel from these libraries were identified. In the next part, we discussed the attention mechanism from Transformers architecture. The standard attention mechanism is bottlenecked by memory bandwidth since intermediate kernels need to read from and write to global memory. FlashAttention2 addressed this issue by fusing all kernels into one using cuTLASS library. It improved the efficiency of attention operation by several magnitudes. This work used TensorCore WMMA API to implement the similar CUDA kernel and explored potential improvements by selecting proper Q,K and V tile sizes. As a result, latency of the FA2 kernel was improved by 10%-40% percent on A100 and RTX3060 GPUs respectively.

Thesis Supervisor: Jurn Gyu Park
Title: Professor

Co-supervisor: Minho Lee
Title: Professor

External Examiner: Zhanat Kappassov
Title: Professor

Acknowledgments

Big thanks to Professor Jurn Gyu Park for constant help, for discipline and for the spirit of a researcher. My sincere gratitude to Nazarbayev University for providing the best conditions for studying and living. And thanks to my family and friends for unconditional love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Deep Learning (DL) methods are very effective in vision and language related tasks [9, 48]. In general, high performing DL models have large sizes. State-of-the art vision models that won ImageNet challenge have high computational demands and even combine Transformer archecture with CNNs [25, 37]. One common application of vision models is related to real-time, such as autonomous robots and surveillance. Large Language Models on the other hand, served in server environment. However training and serving LLMs like LLAMA2 produces several hundreds tons equivalent of CO2 [40]. Even small improvements in efficiency of these models can meaningfully contribute to both the user experience and the environment.

Currently, GPUs are extensively used to train and deploy DL models. Nvidia have produced several generations of GPU, like Turing, Ampere, Hopper and etc, that can be employed for DL model training and serving. They also have provided linear algebra (cuBLAS), deep neural network (cuDNN) specific libraries and inference engines which contain very efficient GPU kernels [5, 26, 29]. In addition to that, users can write their own custom kernels for their need using GPU programming platform and model called CUDA [26].

1.2 Motivation and Contribution

This work consist of two parts. First will focus on CNN models and performance of their implementations using different GPU libraries. Second part addresses issues in the Transformer architecture and current solutions [41]. Also, we develop our own custom CUDA kernel to improve efficiency of Transformer architure and benchmark it against current state-of-the-art implementations.

Chapter 2

Part1. CNN Performance Analysis

2.1 Motivationa and Contribution

CNNs in academia and industry are successfully deployed due to the accuracy improvements in various applications of image classification [18] [36] [14], object detection [4], natural language processing [45], voice recognition [1], etc.

However, one of the most challenging problems is optimizing inference time without significant accuracy degradation. Most recent state-of-the-art models achieve high accuracy at the cost of the performance time [46] [39] especially in computer vision domains, while some fast CNNs experience a significant accuracy degradation as trade-offs [15] considering mobile system domains. One of the recent practical/popular solutions is YOLOv4/VOLOv4-Tiny [4], which focuses on faster object detection, but does not experience a drastic loss in accuracy.

The main point of this work is that CNN inference (and object detection) time is highly dependent on GPU libraries used to implement it and it is possible to accelerate its performance even further by using more optimized ones. Therefore, we adopt and compare three different types of libraries: cuBLAS [26], cuDNN [5], and TensorRT [29]. The cuBLAS library implements basic linear algebra subprograms (BLAS) on top of the NVIDIA CUDA runtime and provides optimized functions/methods for standard CNN operations. While cuDNN GPU-accelerated library primitives are highly tuned for the deep learning domain. TensorRT is an SDK for deep learn-

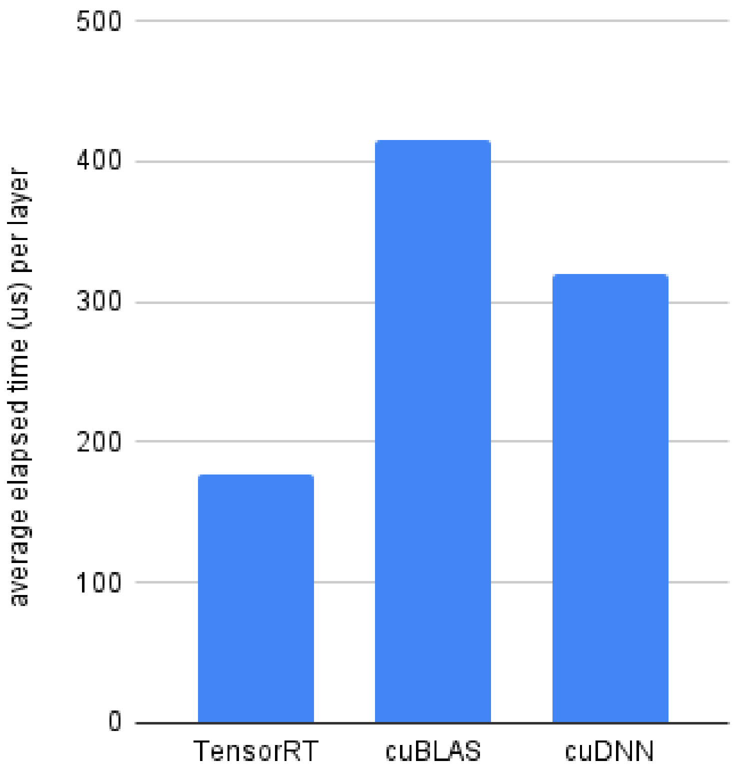


Figure 2-1: Average Elapsed Time (us) per layer.

ing inference with computational graph optimizer by layer fusion and quantization methods.

Considering deep CNN models, notoriously complicated GPU programming implementations, and the hidden property of the libraries, we introduce two combinational approaches for this comparative study: 1) the use of profiling approaches such as *nvprof* [31] or/and *Nsight* [30] and 2) performance evaluation using interpretable ML models, based on the profiled data.

Therefore, this paper makes the following contributions:

- Compare CUDA cuBLAS, cuDNN, and TensorRT Libraries in terms of inference time and GPU kernel usage on YOLOv4-Tiny.

- Propose an interpretable ML-enhanced performance comparison and analysis method using a dataset collected from a profiling tool (*nvprof*).

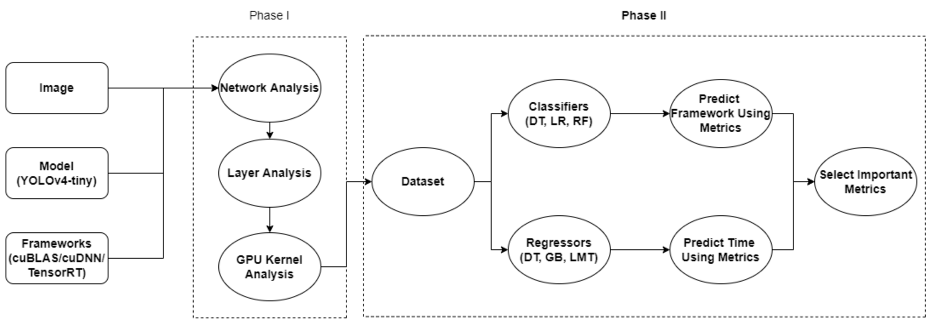


Figure 2-2: Overview of Our Methodology.

- Interpret and explain effects of GPU kernel profiling metrics on execution time using feature importances from the interpretable ML models.

The rest of the paper is organized as follows: Section II describes the motivation and related work. Section III presents our methodology for a systematic comparative study. Section IV shows and analyzes our results. Finally, Section V concludes with future work.

2.2 Related Works

Several works have experimental studies on CNNs implementations with different GPU libraries and frameworks. Wang et al. [43] propose their own framework, which is significantly faster than cuDNN on average, but the source code of their work could not be found. Kim et al. [17] show that cuDNN can improve the performance of different CNN implementations, but they do not compare different GPU libraries. Li et al. [22] provide a comprehensive comparison of different frameworks over a wide range of parameter configurations and investigate potential performance bottlenecks while pointing out a number of opportunities for further optimization. However, there is currently no comparison of different GPU libraries.

Regarding different CNNs with object detection, Felzenszwalb et al. [10] proposed a sliding window approach, where the classifier runs at evenly spaced locations over the entire image. However, this approach is slower and less accurate than YOLO. Girshick et al. [12] introduced a model where R-CNN uses region proposal methods (bounding boxes), but complex pipelines are slow and hard to optimize. Therefore, we adopt YOLOv4-Tiny , as it provides a good trade-off between accuracy and inference time, targeting real-time/embedded systems. Its architecture is given in Appendix (Figure A-1) for a reference.

2.3 Methodology

As shown in Figure 2-2, the workflow of our methodology is divided into two phases: 1) the analysis phase, which is divided into three parts, and 2) the interpretable ML enhanced performance evaluation using the feature importance metrics.

2.3.1 Phase I: Hierarchical Three-step Analysis

The first phase begins with 1) the network analysis step, where overall performance in all layers of three frameworks is compared. We mainly focus on the convolutional layers rather than pooling and fully connected layers, since they are the core part consuming most time in the inference of CNN frameworks. Next, in the second step of 2) layer analysis, representatively important layers are chosen and compared based on the performance time. These usually include layers with the largest and smallest per-layer inference time. And finally, 3) the GPU kernel analysis step is done by analyzing kernels on the chosen representative layers. The analysis of different frameworks is completed through the *nvprof* [31] profiling tool.

2.3.2 Phase II: Interpretable ML enhanced Performance Analysis

By repeating the first phase we capture sufficient data to create a dataset. It includes the name of the library/framework, recorded metrics, and execution time of each iteration. Some implementations of convolution operation launch several consecutive GPU kernels (e.g. cuBLAS launches *im2col* –image to column conversion, followed by *sgemm* –general matrix multiplication). In such cases, we combine metrics from these kernels by weighting them proportionally to their running times. For example, if *im2col* takes 1ms and *sgemm* 4ms, to get the final metrics for the current convolutional layer, we multiply the results of *im2col* by 0.2 and *sgemm* by 0.8 and add them.

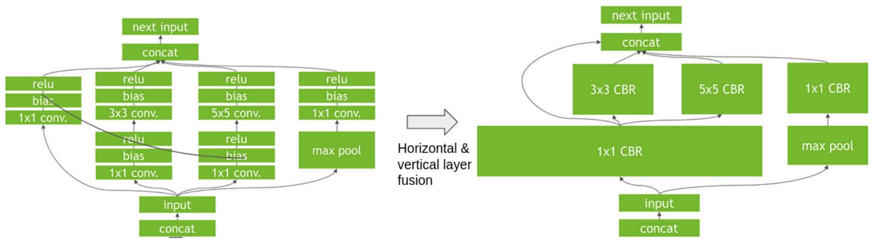


Figure 2-3: Layer Fusion Illustration in TensorRT. [29]

On a completed dataset, three classification models predict the used framework from metrics and three regression ML models predict latency from collected metrics. Frameworks are cuBLAS, cuDNN, or TensorRT. The class labels were transformed to numerical values where class 0 is TensorRT, class 1 is cuBLAS, and class 2 is cuDNN. The algorithms for classification are Decision Tree Classifier (DT), Logistic Regression (LR) and Random Forest Classifier (RFC). For the regression estimators, we use Decision Tree Regressor (DT), Gradient Boosting Regressor (GBR) and Linear Model Tree (LMT) algorithms. To evaluate the performance of the models on the dataset, the Mean Absolute Percentage Error (MAPE) metric is used.

Through the process of training and testing based on these ML models, we are able to select and interpret the most important metrics.

2.4 Results

2.4.1 Experimental Setup

We use GeForce GTX 1050 Ti on Ubuntu 20 with CUDA version 11. The technical specifications are shown in Table 2.4.1.

SM	CUDA cores	Clock rate	Shared Memory	Number of Registers
6	128 per SM	1.39 GHz	48 Kb	65k per block

Table 2.1: Geforce GTX 1050 Ti Specifications. [28]

Dataset

In this experiment, one image [2] was tested for 10 repeated inferences and their performance results were recorded. Single image is used instead of a set of images since the main focus of this paper is the latency of convolutional layers. These results from Phase I are grouped to create the new dataset with profiled (using nvprof) metrics as features. The new dataset consists of 262 samples each having 21 variables. Although the variable names are self-descriptive, the detailed definitions of each variable can be found in Nvidia Profiler User's Guide [31]. Table 2.2 shows the value range of each metric. The framework and performance time is used as target variables for classification and regression models respectively. The results of these interpretable models will be used to analyze and select the most important performance metrics. The collected dataset can be found in [24].

Frameworks

For our experiment, the three frameworks were used to run the model using GPU acceleration: cuBLAS(version 11) [26], cuDNN (version 8.4) [5], and TensorRT (version 8.2) [29]. The Darknet [35] repository provides cuBLAS and cuDNN implementations of the YOLO model, while TensorRT implementation is taken from tkDNN [42] repository.

The cuBLAS is Nvidia's GPU-accelerated implementation for basic linear algebra subprograms (BLAS). The CUDA deep neural network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers.

The NVIDIA TensorRT, achieving state-of-the-art performance, is a high-performance

Feature	Range
warp_execution_efficiency	[1]
gld_efficiency	[0.54; 0.94]
gst_efficiency	[0.57; 1]
stall_inst_fetch	[0.03; 0.12]
stall_exec_dependency	[0.08; 0.32]
stall_memory_dependency	[0.1; 0.5]
stall_sync	[0.08; 0.22]
stall_pipe_busy	[0.01; 0.06]
shared_efficiency	[0.37; 0.77]
achieved_occupancy	[0.25; 0.4]
l2_utilization	[0.13; 0.29]
shared_utilization	[0.3; 0.63]
ldst_fu_utilization	[0.16; 0.27]
cf_fu_utilization	[0.1]
special_fu_utilization	[0.02; 0.16]
tex_fu_utilization	[0.18; 0.22]
single_precision_fu_utilization	[0.21; 0.86]
double_precision_fu_utilization	[0]
dram_utilization	[0.37; 0.89]
type	[0; 2]
time	[259.53; 986.71]

Table 2.2: Dataset Features.

deep learning inference optimizer that delivers low latency and high throughput for inference applications. TensorRT is optimized by kernel auto-tuning, layer and tensor fusion, INT8 and FP16 quantization aware training, and post-training quantization, as shown in Figure 2-3.

Profiling Tools

The *nvprof* [31] profiling tool is NVIDIA's graphical profiler that allows performing a thorough analysis of the model by collecting and viewing profiling data. The visual profiler displays the GPU activity of the application on a timeline including kernel execution, memory transfers, memory set, and CUDA API. The output of profiling can be displayed on the command line or saved in a separate file by *–log-file* command. There are different metrics involved in profiling, such as the number of calls, name of the kernel, minimum, maximum, and average execution times, and either specific metric could be chosen or all metrics could be displayed using *–metrics all* command. To control the profiling and limit it to particular layers, specific instructions named *cudaProfilerStart()* and *cudaProfilerStop()* are used to indicate the start and the end of the profiling.

For the second phase (Phase II), the analysis results will be grouped to form a dataset with the execution time of each kernel, framework types, and total running time. The dataset set will be used in two tasks: regression and classification. In the regression tasks, execution time is the predicted value, while framework type is the label class for classification.

2.4.2 Results and Analysis

Network analysis

To perform network layer analysis, the execution times of all convolutional layers are compared for each framework. The execution time for all layers is obtained through the nvprof profiling tool and only results from convolutional layers are selected. As depicted in Figure 2-4, TensorRT (with both FP32 and INT8 implementations) is the fastest along all layers, while cuDNN is in the middle and cuBLAS is the slowest. However, despite being the fastest one, the INT8 version of TensorRT results in a significant accuracy drop.

It can be seen that some layers take more time to execute while others take less, for example, the 3rd and 20th convolutional layers are slow. Even though Figure 2-4 provides some information on elapsed time for each layer, it doesn't tell why the particular layers are slower, and in order to check the reasons behind such results, each layer should be analyzed independently.

Layer analysis

Convolutional layers have different execution times, which can be caused by kernels used in the framework, and by the architecture of the network itself. According to Table 2.3, all the slowest layers have 3x3 filters, while fast layers mostly have 1x1

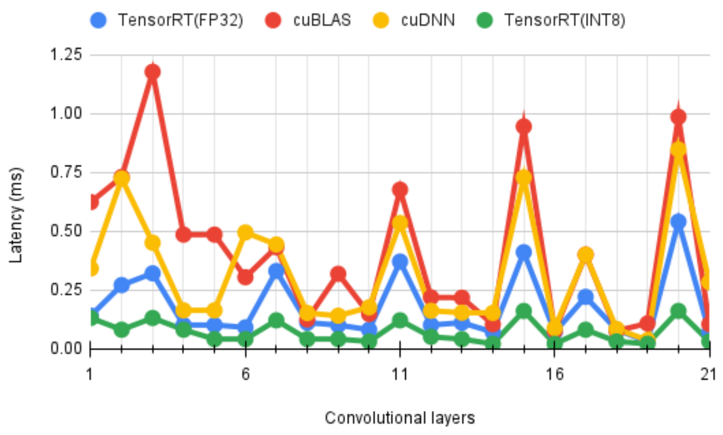


Figure 2-4: Elapsed time for convolutional layers.

Number of Input Filters		Filter Size: 3x3	Filter Size: 1x1
	< 128	1,4,5,8,9,2,3	6
	>= 128	7,12,13,17, 11,15,20	10,14,16,18,19,21

Table 2.3: Layers with Fast(top5), Slow(bottom5), and Intermediate(rest) latencies according to cuBLAS results.

ones. Also, faster layers tend to have more than 128 output filters as well. Slow layers can have both more and less than 128 filters. While this does not mean that all layers with 3x3 filters will be slow, in combination with the kernels these layers use, this factor could be influential. Convolutional layers that are considered in this work are 3 and 20, their architectures are given in Table 2.4

GPU kernel analysis

The convolutional layers 3 and 20 are the slowest in execution time, and it is necessary to check which kernels are used and compare their performances. Figure 2-5 shows the third convolutional layer's kernels for each framework. cuDNN and TensorRT each launch single kernel (*maxwell_scudnn_winog*

layer#	filters	filters/stride/pad	input	output
3	64	3x3/1/1	104x104x64	104x104x64
20	256	3x3/1/1	26x26x384	26x26x256

Table 2.4: Structures of convolutional layers 3 and 20.

Model	Test MAPE	Important Feature 1	Important Feature 2	Important Feature 3
Decision Tree	0.0094	gld_efficiency	shared_utilization	shared_efficiency
Gradient Boosting	0.0088	gld_efficiency	stall_memory_dependency	achieved_occupancy
Linear Model Tree	0.0079	stall_exec_dependency	stall_sync	gld_efficiency

Table 2.5: Performance of Regression Models.

Model	Test Accuracy	Important Feature 1	Important Feature 2	Important Feature 3
Decision Tree	100%	gld_efficiency	achieved_occupancy	-
Logistic Regression	100%	gld_efficiency	shared_efficiency	shared_utilization
Random Forest	100%	ldst_fu_utilization	stall_exec_dependency	gld_efficiency

Table 2.6: Performance of Classification Models.

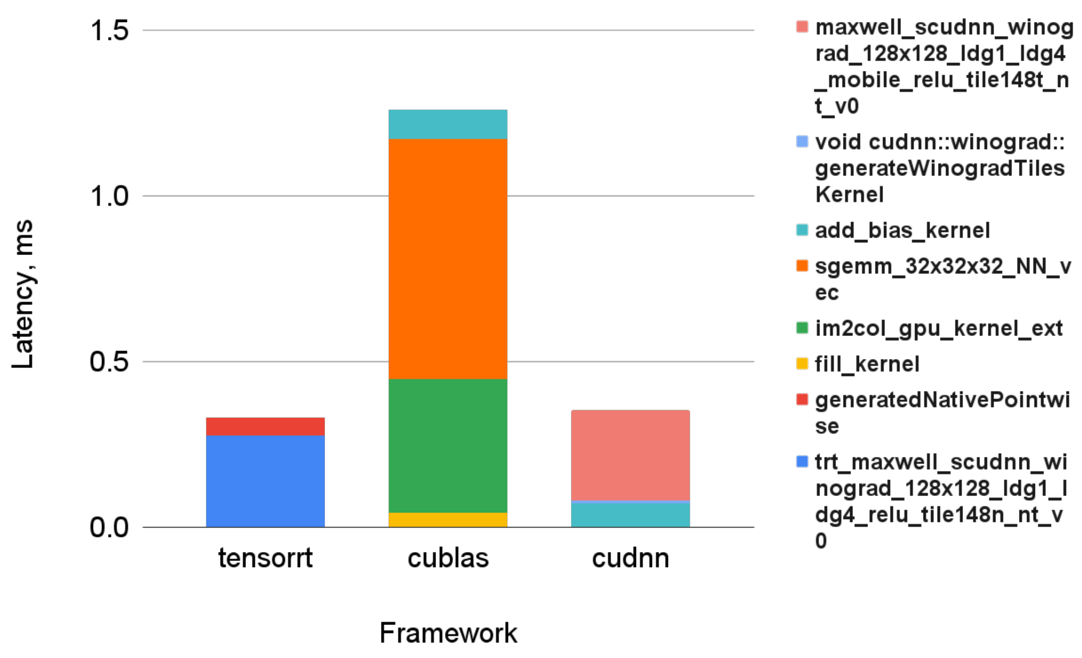


Figure 2-5: GPU kernels in layer 3.

rad_128x128_ldg1_ldg4_mobile_relu_tile148t_nt_v0 and *trt_maxwell _scudnn_winograd_128x128_ldg1_ldg4_relu_tile148n_nt_v0* accordingly) to perform both convolution and ReLU operations. By speculating kernel names, since it is a closed-source implementation, they are based on Winograd's minimal filtering algorithm [44]. The algorithm reduces the number of multiplications by using extra addition operations. It is the most efficient convolution implementation when the filter size is 3x3 and the batch size is small [17]. In the case of convolutional layer 20, the filter size is 3x3 and the batch size is 1. cuBLAS, on the other hand, launches *fill_kernel, im2col_gpu_kernel_ext*, and *sgemm_32x32x32_NN_vec* kernels to perform convolution operations. The first two GPU kernels are designed to reduce convolution operation to matrix multiplication and solely memory bandwidth bounded. SGEMM kernel does matrix multiplication and it utilizes both memory and compute units in balance. In contrast, Winograd's minimal filtering-based kernels consume more static shared memory and registers.

Dataset

Before performing any classification or regression, the dataset was modified first. According to Table 2.2, three features do not change their value throughout the dataset (*warp_execution_efficiency*, *cf_fu_utilization*, and *double_precision_fu_utilization*). Therefore, they were removed from the model building, as they did not provide any essential information, and this could result in more noise. Also, the time values were divided by 1000, so they could be in the range [0; 1].

Regression

The regression task focused on predicting the execution time of each iteration. Table 2.5 shows that the average MAPE is less than 0.01 for each model. The Linear Model Tree shows a slightly lower result. The feature importance was calculated using scikit-learn's built-in Gini importance metric [33]. *gld_efficiency* appears as one of the most important metrics in all of the interpretable models used. Other metrics appear only once, and their importance is usually significantly lower. Therefore, it can be concluded that according to the regression models, *gld_efficiency* is one of the most important metrics as shown in Figure 2-6.

Classification

Classification models predict the used framework based on the provided metrics. Each model has 100% accuracy on the used dataset, which is shown in Table 2.6. Such high accuracy is due to the small size of the dataset, which uses only 3 classes for classification, and has a low level of noise. Similar to the regression, the feature

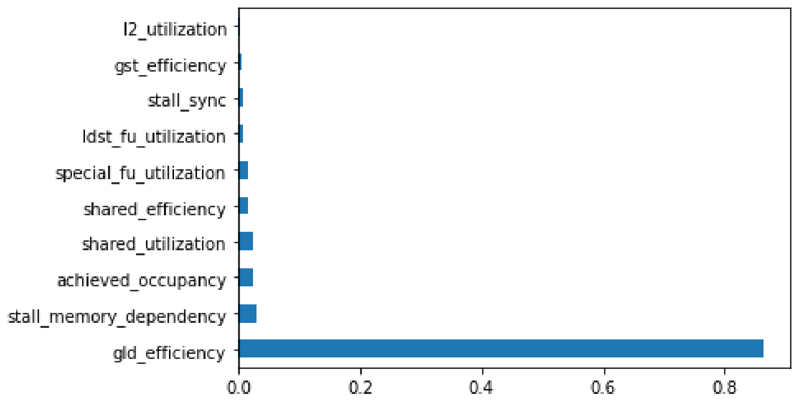


Figure 2-6: Gradient Boosting Regression feature weights.

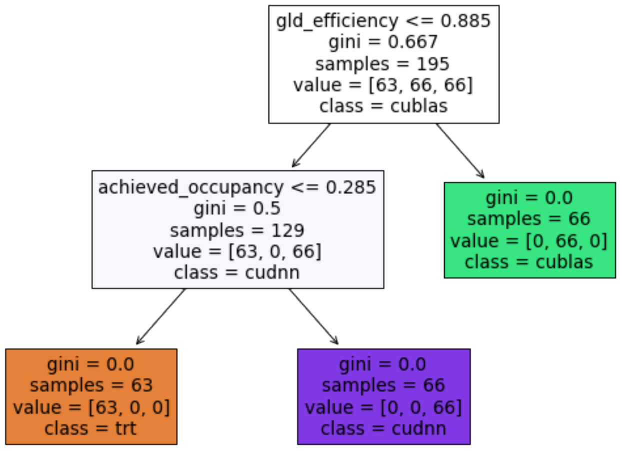


Figure 2-7: Decision Tree Classification.

importance is calculated using scikit-learn's calculations of the Gini importance as shown in Figure 2-7. In Logistic Regression, however, we use the coefficients of the features in the decision function, as this model does not have feature importance by itself. Our interpretation is that the larger the absolute value of the coefficient, the more important the feature is. Again, *gld_ efficiency* is the most important metric in two out of three classifiers, followed by *ldst_fu_ utilization*. These features, as well as *shared_ utilization*, *shared_ efficiency*, and *stall_ exec_ dependency* appear in both classifiers and regressors.

Chapter 3

Part2. Attention Optimizations on GPU

3.1 Motivation and Contribution

GPU is the main accelerator hardware that is being used to train and serve deep learning models. The main advantages of GPU over other hardware are high memory and compute bandwidth and versatility. Current state-of-the-art Large Language Models (LLM) have up to 540 billion parameters [48]. For example, to train LLAMA2 model almost 3 million A100 GPU hours were spent [40].

The Scaled Dot Product Attention (SDPA) introduced by Waswani et. al is the key building block of all these LLMs [41] and it is formulated as below:

$$SDPA(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d}})V \tag{3.1}$$

where Q, K, V are matrices with rows each representing the embedding of a particular token. Usually, embedding size is upto 1024 while the number of tokens can be as large as 100K. This means that temporary matrix QK^T can have 100K by 100K dimensions. For this reason Attention has quadratic memory and runtime complexity in terms of number of tokens. Memory bandwidth becomes the bottleneck while under-utilizing the compute units of a GPU.

Instead of performing *matrix-matrix multiplication* (matmul), then $softmax$ and then again *matmul*, the single GPU kernel fusing all these operations into one could solve the above issue. Kernel fusion is among the widely used techniques in libraris such as cuDNN and TensorRT [5, 29].

FlashAttention (FA1) and its improvement FlashAttention2 (FA2) employ the kernel fusion technique on SDPA and provide the pytorch compatable, IO-aware, fused GPU kernels for both forward and backward passes [8, 7]. FA1 is $x7.6$ faster than default pytorch implementation. FA2 uses cuTLASS library to utilize TensorCores and it is $x2$ faster than its predecessor [8, 7].

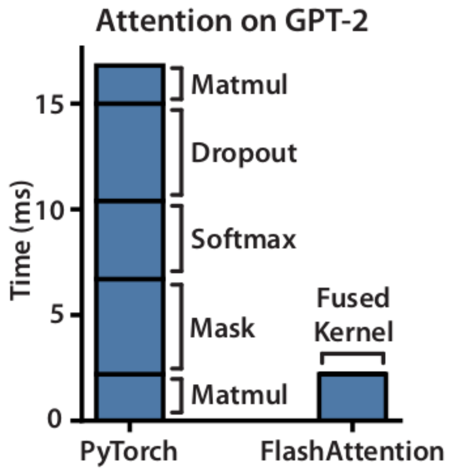


Figure 3-1: Speedup over pytorch implementation [8]

FA2 is optimized for A100 GPU and uses cuTLASS to utilize TensorCores. WMMA API has also been succesfully used in many DL TensorCore optimizations recently [11, 47, 13]. This is the main point we considered as a motivaton to our work:

- Apply further optimizations for consumer-level GPUs like RTX series by selecting proper Q,K,V tile sizes using WMMA API

In this work, we implement a very similar fused kernel for SDPA using CUDA Tensor Cores WMMA API for Ampere (SM80) and go through all optimizations techniques used [26].

Our contributions are here:

- Implement SDPA using TensorCore WMMA API
- Explore effects of tiles sizes for Q, K and V matrices
- Improve the latency of SDPA kernel by 10%-40% with respect to FA2 in Ampere based GPUs.
- All the codes are available at: *https://github.com/zhumakhan/flash-attention-wmma.git*

3.2 Related Works

Rabe et al [34] observed that SDPA does not need $O(n^2)$ memory. They replaced softmax operation with *lazy* softmax [16]. Given query vector q, key and value matrices K and V (scaling part is omitted):

$$s_i = dot(q, K_i), \quad s_i^* = e^{s_i}, \quad attention(q, K, V) = \frac{\sum_i V_i s_i^*}{\sum_j s_j^*} \tag{3.2}$$

With this formulation, calculating only a chunk of s_i^* at a time is sufficient to calculate the full attention. Jax and Pytorch implementations of the algorithm that use $O(\sqrt{n})$ memory are also provided in this work. Following this, Xformers library developed a specialized GPU kernel to avoid intermediary memory transfers and kernel launch overheads [21]. This is known as a $memory\ efficient\ attention$ in Pytorch framework.

PagedAttention is introduced as an efficient attention inference operation that avoids GPU memory fragmentation by keeping key-value cache as a separate list of vectors instead of a matrix [20]. In standard inference more than 30% of DRAM is filled by key-value cache and at each iteration it needs to be recomputed. However, the only difference between the key-value cache of current iteration and previous iteration is an additional column vector. PagedAttention avoids re-computation by preserving previous values and adding a new vector to the list. vLLM is an inference engine built on top of PagedAttention [19]. It increased the throughput by $x2$-4 without increasing the latency with respect to state-of-the-art systems like FasterTransformer and Orca.

FlashAttention is a GPU kernel for SDPA and it also uses *online lazy* softmax [8, 23]. Within a single thread-block, in the outer loop tiles of K and V are loaded into shared memory and in the inner loop a tile of Q is loaded. Then partial SDPA calculated and stored in global memory. The number of heads in multi-head attention and number of batches define the grid size of the kernel.

FlashAttention2 is the improved version of FlashAttention algorithm [7]. Inner and outer loops are swapped. Outer loops is parallelized over multiple thread blocks. It is implemented using the cuTLASS library to utilize the TensorCore units that are available in new generations of GPU like Ampere [26]. As a result, it doubled the speed.

There are several GPU compilers that can to certain extent fuse operations into single GPU kernel. OpenAI/Triton implementation of FA2 has comparable speed with FA1 and Xformers' implementation of memory efficient attention [38]

Colflax is an implementation of FA2 using cuTLASS library for Hopper (SM90) architecture. It utilizes Hopper specific features such as Tensor Memory Accelerator (TMA) and Warpgroup Matrix-Matrix-Accumulate (WGMMA). It also explores and selects optimal tile size for Q, K and V matrices balancing register pressure and shared memory utilization. It achieved 20%-50% speedup over FA2 which is optimized for Ampere (SM80) [3].

ByteTransformers is another work that implements fused kernel for SDPA [47]. Number of tokens or context length could have varying sizes within the batch and common solution is to pad to make them all have the same length. ByteTransformers address that issue by implementing a CUDA kernel that accepts varying size inputs.

Current state-of-the-art SDPA implementation on Ampere is FA2 and on Hopper is Colflax.

	HW	Core	Software	QKV tile	Speed
mem eff. [34]	multi	Tensor	cuTLASS	n/a	slow
FA1 [8]	multi	Cuda	plain CUDA	n/a	slow
FA2 [7]	Ampere+	Tensor	cuTLASS	32x128	fast
Triton [38]	Hopper+	Tensor	cuTLASS	n/a	slow
Colflax [3]	Hopper+	Tensor	cuTLASS	64x128	fast
Byte Transfor mers [47]	Apmere+	Tensor	cuTLASS WMMA API	16x64	slow
Ours	Ampere	Tensor	WMMA API	32x64	fast

Table 3.1: Fused GPU kernel implementations for SDPA.

3.3 Methodology

In this section, we discuss details of the algorithm and effects of Q, K and V tile sizes on performance. Overall methodology diagram is given in Figure 3-2.

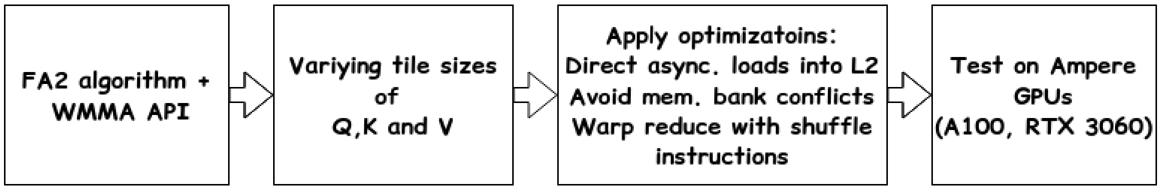

FA2 algorithm + WMMA API
Variying tile sizes of Q,K and V
Apply optimizatoins: Direct async. loads into L2 Avoid mem. bank conflicts Warp reduce with shuffle instructions
Test on Ampere GPUs (A100, RTX 3060)

Figure 3-2: Methodology diagram.

3.3.1 Algorithm and Design

The main difference between FA1 and FA2 is that outer and inner loops are swapped and FA2 utilizes TensorCores. We use the same algorithm as FA2 [7] and its diagram is given in Figure 3-3.

Adopted FlashAttention2 algorithm [7]

```
Given Q, K, V: $\in \mathbb{R}^{N\times d}$ matrices in HBM, block sizes $B_c$, $B_r$

 1. Partition Q into $T_r$=$\frac{N}{B_r}$ blocks $Q_1$,...,$Q_{T_r}$ of size $B_r \times d$ each. Divide K,V into
    $T_c = \frac{N}{B_c}$ blocks $K_1$,...,$K_{T_c}$ and $V_1$,...,$V_{T_c}$, of size $B_c \times d$ each.
 2. **for** 1 $\leq$ i $\leq$ $T_r$ do
 3.     Load $Q_i$ from HBM to on-chip SRAM
 4.     On chip, initialize $O_i^{(0)}$=$(0)_{B_r\times d} \in \mathbb{R}^{B_r\times d}$,
        $l_i^{(0)} = (0)_{B_r} \in \mathbb{R}^{B_r}$, $m_i^{(0)} = (-\infty)_{B_r} \in \mathbb{R}^{B_r}$
 5.     **for** 1 $\leq$ j $\leq$ $T_c$ do
 6.         Load $K_j, V_j$ from HBM to on-chip SRAM
 7.         On chip, compute $S_i^{(j)} = Q_i K_j^T \in \mathbb{R}^{B_r\times B_c}$
 8.         On chip, compute $m_i^{(j)}$=max($m_i^{(j-1)}$, rowmax($S_i^j$)) $\in \mathbb{R}^{B_r}$,
            $P_i^{(j)}$=exp($S_i^{(j)} - m_i^{(j)}$)$\in \mathbb{R}^{B_r\times B_c}$,
            $l_i^{(j)}$=$e^{m_i^{(j-1)}-m_i^{(j)}} l_i^{(j-1)}$ + rowsum($P_i^j$) $\in \mathbb{R}^{B_r}$
 9.         On chip, compute $O_i^{(j)}$=diag($e^{m_i^{(j-1)}-m_i^{(j)}}$)$^{-1}O_i^{(j-1)}$+$P_i^{(j)}V_j$.
10.     **end for**
11.     On chip, compute $O_i$=diag($l_i^{(T_c)}$)$^{-1}O_i^{(T_c)}$
12.     Write $O_i$ to HBM as $i$-th block of O.
13. **end for**
14. Return the output O.
```

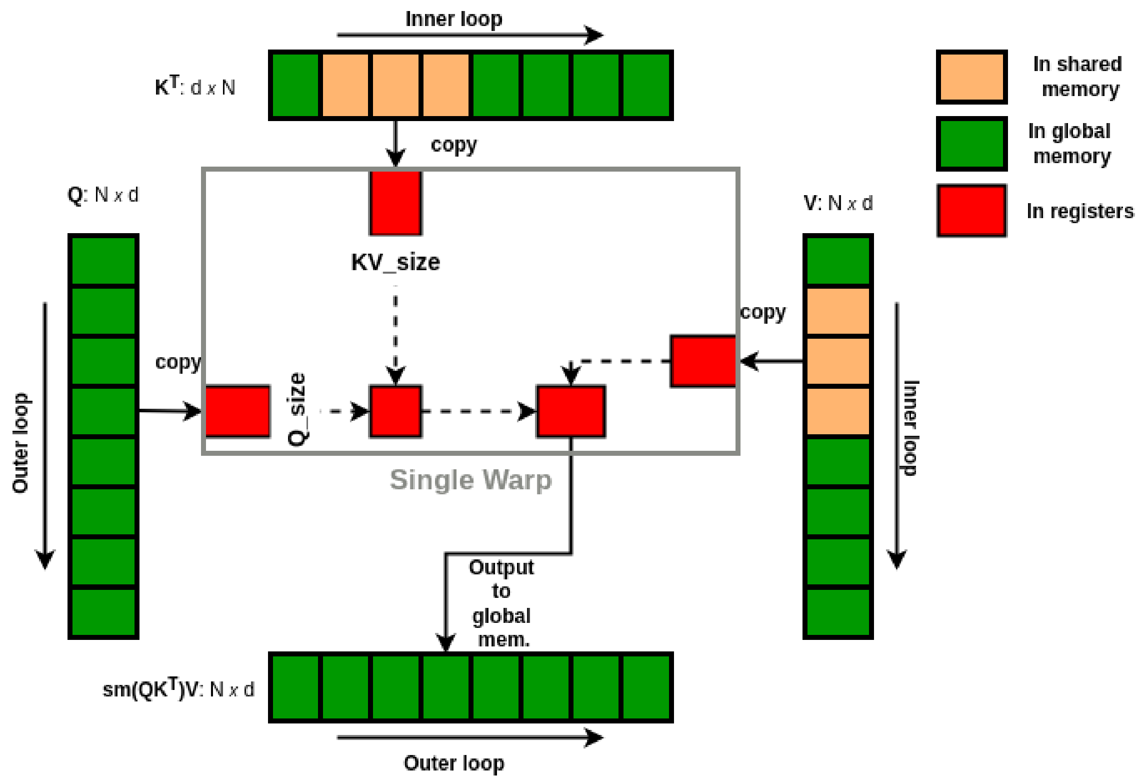


Figure 3-3: Visual diagram of the algorithm.

Each iteration in outer loop is matched to a single thread block of CUDA. Inner loop is parallelized using the threads within the single thread block. On chip SRAM is a register file in our implementation. Tiles of K and V are loaded into shared memory first, then copied into registers to utilize TensorCore.

In next sections, a background on optimization techniques and effects of tiles on number of registers are given.

3.3.2 Async vectorized loads bypassing register

Prior to Ampere architecture, loading from HBM to shared memory was a two-step process: first data is loaded into registers, then to shared memory. Starting from Ampere, memory loads can bypass registers and directly go to shared memory as shown in Figure 3-4. This can be achieved by using the asynchronous data movement instructions. We used inlined *cp.async* PTX instruction [6]. *cp.async.cg* instruction indicates caching at L2 level and not at L1 level. Whereas *cp.async.ca* instruction caches a data at all levels up to the L1 cache.

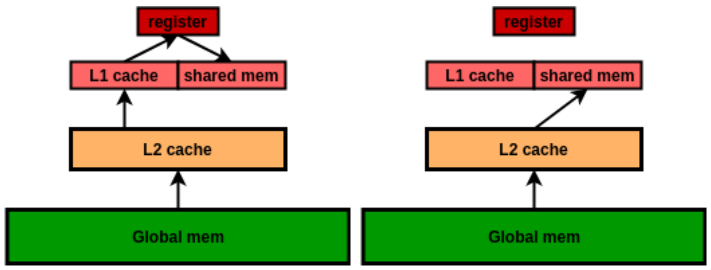
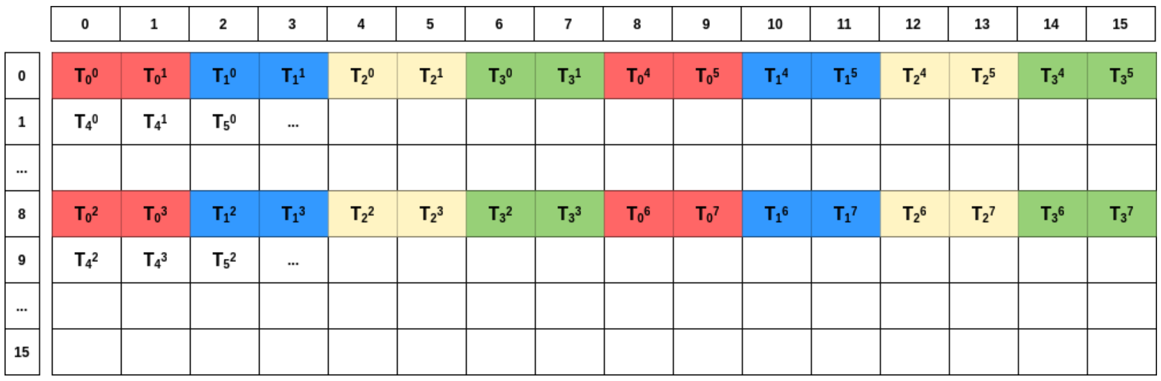


Figure 3-4: Global to shared data movement prior (on the left) and after (one the right) Ampere.

3.3.3 Padding to avoid shared memory bank conflicts

The mapping between row-major 2-D matrix and WMMA API's *fragment* matrix defines the padding size. All the elements of a fragment are stored in registers and spread accross the threads of a warp. Also, this is architecture specific. Below is the mapping for Ampere architecture:



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	T_0^0	T_0^1	T_1^0	T_1^1	T_2^0	T_2^1	T_3^0	T_3^1	T_0^4	T_0^5	T_1^4	T_1^5	T_2^4	T_2^5	T_3^4	T_3^5
1	T_4^0	T_4^1	T_5^0	...												
...																
8	T_0^2	T_0^3	T_1^2	T_1^3	T_2^2	T_2^3	T_3^2	T_3^3	T_0^6	T_0^7	T_1^6	T_1^7	T_2^6	T_2^7	T_3^6	T_3^7
9	T_4^2	T_4^3	T_5^2	...												
...																
15																

Figure 3-5: Mapping of 16x16 matrix into wmma::fragment in Ampere. T_i^j means j-th element of i-th thread in a warp.

In the mapping above, first four threads of a warp collectively store two rows (0-th and 8-th) of a matrix. Next four threads store 1-st and 9-th rows, and so on. Thus, to store 16 such rows 32 threads are enough.

Given any $64x64$ matrix A, it can be partitioned into $16x16$ matrices to be able to use wmma fragments. Two adjacent elements of a matrix are stored in a single shared memory bank, because elements should have $half$ (16 bits) datatype while one bank is 32 bits wide.

If matrix A is stored in $64x64$ shared memory, then rows of any of $16x16$ sub-matrices fall into the same 8 banks and each of its columns entirely fall into a single bank. As illustrated in Figure 3-5, 4 different threads access the row while 8 different threads access the column of $16x16$ matrix when loading from shared memory into a fragment. This, in theory, causes 8-way shared memory bank conflict.

If A is stored in $64x72$ shared memory with the padding of 8 elements at the end of each row, we end up with row-to-bank mapping as in Table 3.2 below:

row	0	1	2	3	4	5	6	7	8	...
bank	0-7	4-11	8-15	12-19	16-23	20-27	24-31	28-3	0-7	...

Table 3.2: Mapping of rows of 16x16 matrix into banks of shared memory after padding.

This padding size totally eliminates the bank conflicts when loading 16x16 matrix from shared memory into wmma fragments. Similar technique can be applied for different shapes of matrices as well.

3.3.4 Row-max and -sum in registers using shuffle instructions

Referring to Figure 3-5, each row is entirely stored in 4 threads collectively. For example, to find the max value from the first row, max value of each thread is calculated first. Thus, maximum of red cells are stored in thread 0, maximum of blue cells are stored in thread 1 and so on. Then, $shuffle$ instruction is used to communicate between threads of a warp [27]:

```
$for(intj = 2; j > 0; j = j/2)$
    $max\_val = max(max\_val,\ \_\_shfl\_xor\_sync(uint32\_t(-1), max\_val, j))$
```

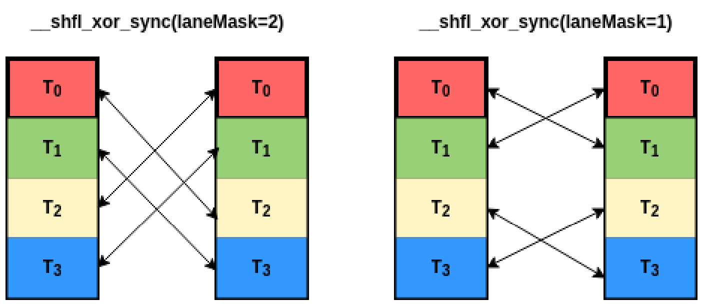


Figure 3-6: Finding maximum across 4 threads

The first iteration of above loop is depicted on the left and second iteration on the right of Figure 3-6. Similarly, row-sum can be computed within the warp.

3.3.5 Effects of Q_size and KV_size

In the algorithm in Figure 3-3, increasing Q_size and KV_size increase the register pressure on the kernel. In contrary, increasing Q_size will reduce the number of thread blocks to launch. While KV_size acts as a tile size to copy from shared memory into registers.

Q_size	KV_size	32	64	128	160
16		80	91	106	116
32		128	148	192	202
64		217	255	255	255

Table 3.3: Number of register used in the kernel. Red color means a register spill.

In the experiments, the latency will be the main measurement criteria. In addition to it, GPU performance metrics will be observed and discussed in relation to tile sizes Q_size and KV_size.

3.4 Results

3.4.1 Experimental Setup

These are the compiler flags used:

-Xptxas -O3,-v -maxrregcount=255 –expt-relaxed-constexpr -fmad=true -ftz=true -prec-div=false -prec-sqrt=false

Name	SM	Cuda Cores	Tensor Cores	SM freq.	Shared Memory	Number of Registers	HBM
A100	108	128 per SM	4 per SM	1.09 GHz	164 Kb	65k	40GiB
RTX3060	30	128 per SM	4 per SM	0.816 GHz	48 Kb	65k	6GiB

Table 3.4: GPU specifications

3.4.2 Results and Analysis

In GPUs with TensorCore, Bytes-to-Flops ratio is small [32]. Compute throughput of our kernel with all different tile sizes is less 70%, while it is trivial to achieve full memory bandwidth utilization. To put simply, memory bandwidth is a bottleneck in most cases.

Decreasing Q_size increases the grid dimension, hence puts more pressure on L2 cache per SM. This reduces the global memory throughput as shown in Figure 3-7. In case of Q_size=64, high register pressure and register spill causes very high global memory throughput. Q_size=32 and KV_size={64,128} achieve balanced global memory throughput and have low latency (Table 3.5).

Q_size KV_size	32	64	128	160
16	222.01	220.81	222.49	235.94
32	198.58	**195.15**	196.7	202.45
64	200.47	226.66	322.45	587.05

Table 3.5: Latency (ms) of our kernel for batch size=4 #heads=12, #tokens=25K and head dimension=64 on A100(SXM4-40GB) GPU.

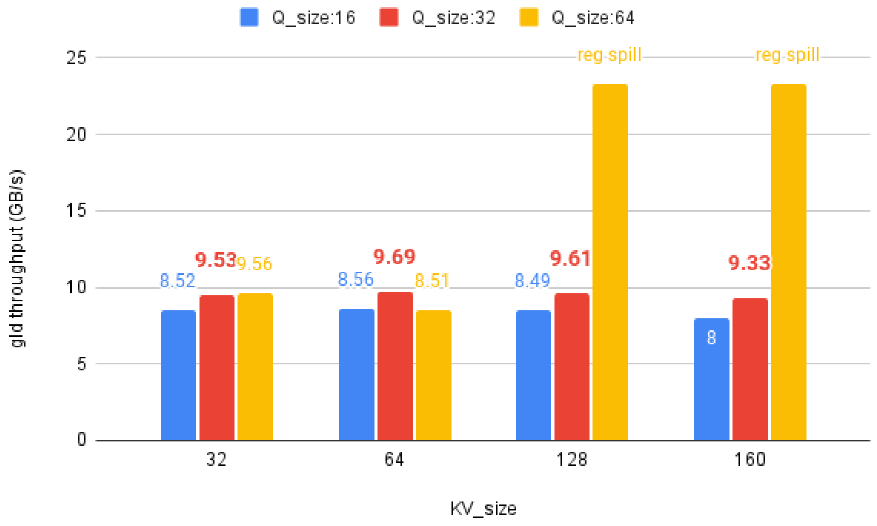


Figure 3-7: Global memory throughput.

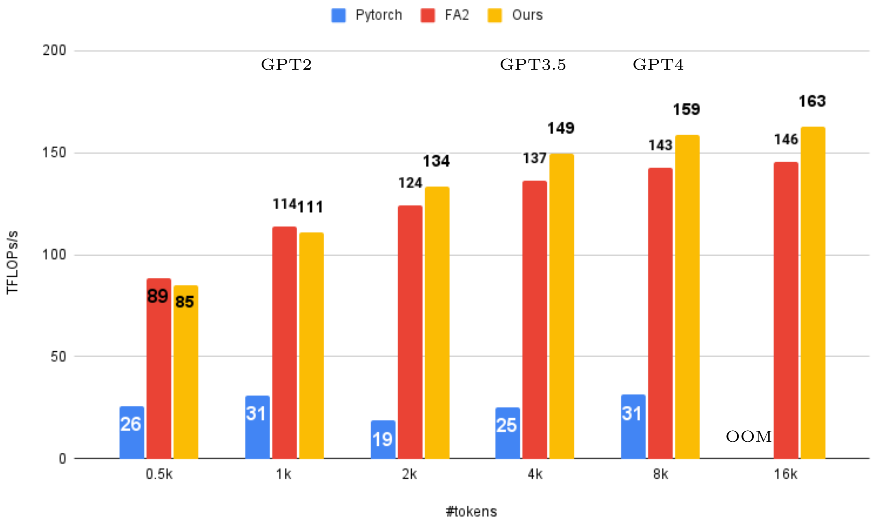


Figure 3-8: Speed (TFLOPs/s) of our kernel for batch size=4, #heads=12, #tokens, and head dimension=64 on A100(SXM4-40GB)GPU.

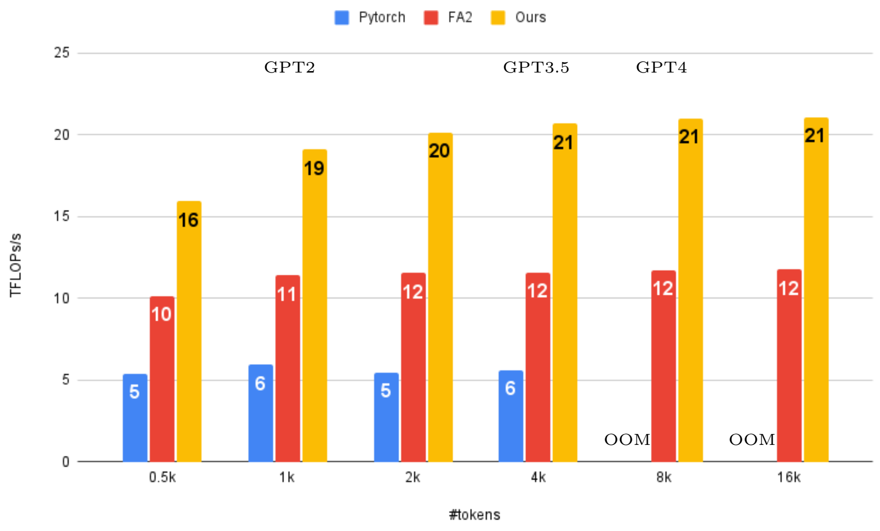


Figure 3-9: Speed (TFLOPs/s) of our kernel for batch size=4, #heads=12, #tokens, and head dimension=64 on laptop RTX3060 GPU.

Further, we use kernel with Q_size=32 and KV_size=64 configurations to compare with FA2. On A100 GPU, our kernel achieves more than 10% speedup over FA2 (Figure 3-8). As number of tokens increase, difference gets larger. On our laptop RTX 3060 GPU, inference speedup improvement becomes 40% of FA2 (Figure 3-9). This is because FA2 was solely optimized for A100 GPU and our kernel has more flexibility.

Chapter 4

Conclusion

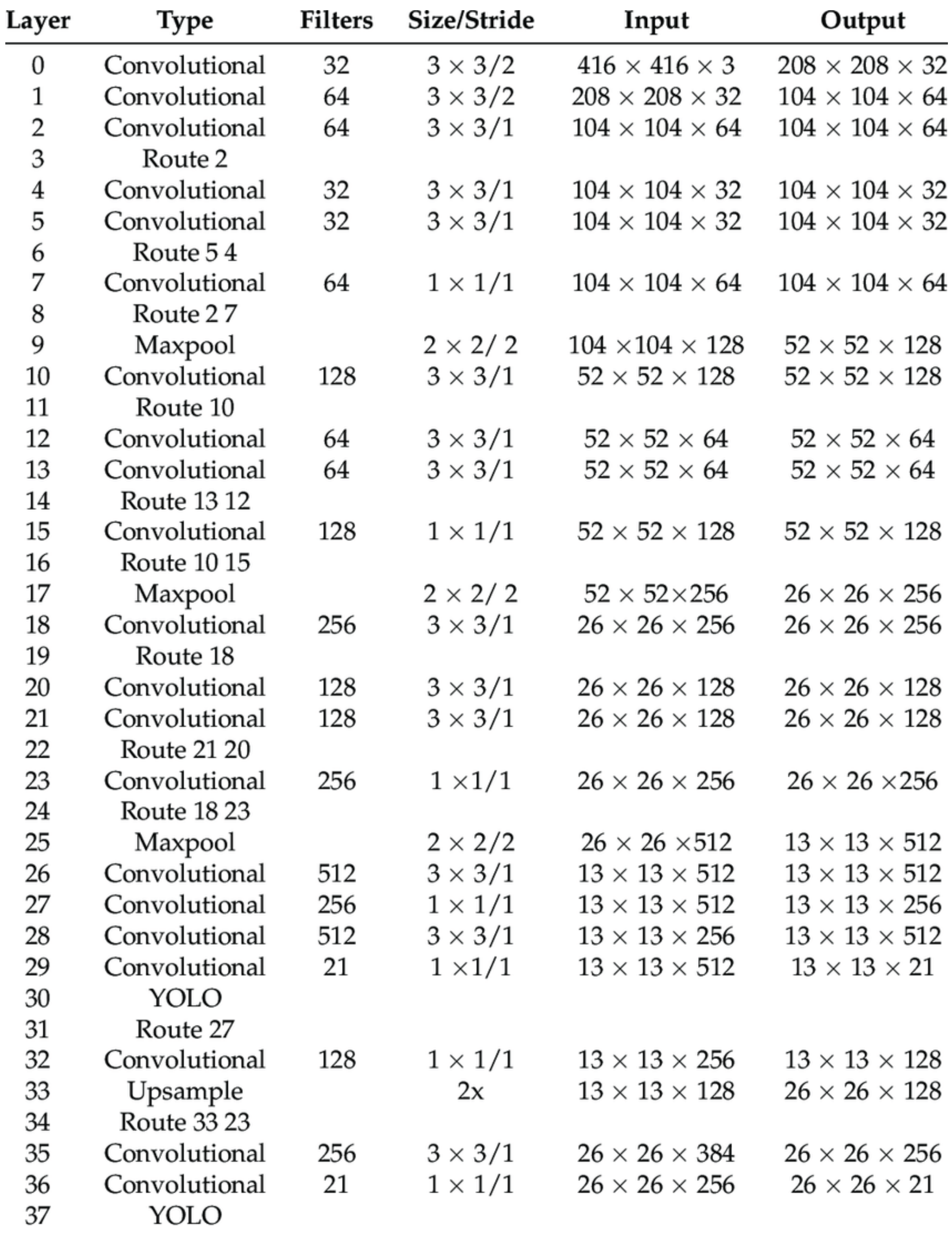
We have comprehensively investigated and compared three different GPU frameworks, which improve the performance of CNNs. Applying these frameworks to the YOLOv4-Tiny shows that TensorRT results in the most noticeable reduction in latency.

Using interpretable ML models in the classification and regression tasks shows that *gld_ efficiency* and *ldst_fu_ utilization* provide the most information, and are the most important metrics as a result. This deduction is supported by the fact that all the used ML models show 100% accuracy in the classification and 0.0094 MAPE in the regression tasks respectively.

Also, we have implemented FA2 GPU kernel using WMMA API and achieved 10% and 40% speedups on A100 and RTX3060 GPUs respectively. We have also observed the effects of tile sizes, Q_size and KV_size, on memory throughput and latency. Moreover, detailed optimizations techniques employing TensorCores is covered. Currently our kernel does not support causal masking and head dimension is limited to 64. These are considered for a future work.

Appendix A

Figures



Layer	Type	Filters	Size/Stride	Input	Output
0	Convolutional	32	3 × 3/2	416 × 416 × 3	208 × 208 × 32
1	Convolutional	64	3 × 3/2	208 × 208 × 32	104 × 104 × 64
2	Convolutional	64	3 × 3/1	104 × 104 × 64	104 × 104 × 64
3	Route 2				
4	Convolutional	32	3 × 3/1	104 × 104 × 32	104 × 104 × 32
5	Convolutional	32	3 × 3/1	104 × 104 × 32	104 × 104 × 32
6	Route 5 4				
7	Convolutional	64	1 × 1/1	104 × 104 × 64	104 × 104 × 64
8	Route 2 7				
9	Maxpool		2 × 2/ 2	104 ×104 × 128	52 × 52 × 128
10	Convolutional	128	3 × 3/1	52 × 52 × 128	52 × 52 × 128
11	Route 10				
12	Convolutional	64	3 × 3/1	52 × 52 × 64	52 × 52 × 64
13	Convolutional	64	3 × 3/1	52 × 52 × 64	52 × 52 × 64
14	Route 13 12				
15	Convolutional	128	1 × 1/1	52 × 52 × 128	52 × 52 × 128
16	Route 10 15				
17	Maxpool		2 × 2/ 2	52 × 52×256	26 × 26 × 256
18	Convolutional	256	3 × 3/1	26 × 26 × 256	26 × 26 × 256
19	Route 18				
20	Convolutional	128	3 × 3/1	26 × 26 × 128	26 × 26 × 128
21	Convolutional	128	3 × 3/1	26 × 26 × 128	26 × 26 × 128
22	Route 21 20				
23	Convolutional	256	1 ×1/1	26 × 26 × 256	26 × 26 ×256
24	Route 18 23				
25	Maxpool		2 × 2/2	26 × 26 ×512	13 × 13 × 512
26	Convolutional	512	3 × 3/1	13 × 13 × 512	13 × 13 × 512
27	Convolutional	256	1 × 1/1	13 × 13 × 512	13 × 13 × 256
28	Convolutional	512	3 × 3/1	13 × 13 × 256	13 × 13 × 512
29	Convolutional	21	1 ×1/1	13 × 13 × 512	13 × 13 × 21
30	YOLO				
31	Route 27				
32	Convolutional	128	1 × 1/1	13 × 13 × 256	13 × 13 × 128
33	Upsample		2x	13 × 13 × 128	26 × 26 × 128
34	Route 33 23				
35	Convolutional	256	3 × 3/1	26 × 26 × 384	26 × 26 × 256
36	Convolutional	21	1 × 1/1	26 × 26 × 256	26 × 26 × 21
37	YOLO				

Figure A-1: Yolov4-tiny architecture

Bibliography

[1] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing*, 22(10):1533–1545, 2014.

[2] Alexeyab. Test image used for performance comparison. `https://github.com/AlexeyAB/darknet/blob/master/data/dog.jpg`, Sep. 2, 2016.

[3] Ganesh Bikshandi and Jay Shah. A case study in cuda kernel fusion: Implementing flashattention-2 on nvidia hopper architecture using the cutlass library. *arXiv preprint arXiv:2312.11918*, 2023.

[4] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[6] NVIDIA Compute. Ptx: Parallel thread execution isa version 2.3. *Dostopno na: http://developer. download. nvidia. com/compute/cuda*, 3:1–203, 2010.

[7] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[9] Shi Dong, Ping Wang, and Khushnood Abbas. A survey on deep learning and its applications. *Computer Science Review*, 40:100379, 2021.

[10] Pedro F Felzenszwalb, Ross B Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE transactions on pattern analysis and machine intelligence*, 32(9):1627–1645, 2010.

[11] Daniel Y. Fu, Hermann Kumbong, Eric Nguyen, and Christopher Ré. FlashFFTConv: Efficient convolutions for long sequences with tensor cores. 2023.

[12] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[13] Mohammad Hafezan and Ehsan Atoofian. Improving energy-efficiency of capsule networks on modern gpus. *IEEE Computer Architecture Letters*, 2024.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[15] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[16] Hanhwi Jang, Joonsung Kim, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. Mnnfast: A fast and scalable system architecture for memory-augmented neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 250–263, 2019.

[17] Heehoon Kim, Hyoungwook Nam, Wookeun Jung, and Jaejin Lee. Performance analysis of cnn frameworks for gpus. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 55–64. IEEE, 2017.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[19] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Yu, Joey Gonzalez, Hao Zhang, and Ion Stoica. vllm: Easy, fast, and cheap llm serving with pagedattention, 2023.

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[21] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. xformers: A modular and hackable transformer modelling library. `https://github.com/facebookresearch/xformers`, 2022.

[22] Xiaqing Li, Guangyan Zhang, H Howie Huang, Zhufan Wang, and Weimin Zheng. Performance analysis of gpu-based convolutional neural networks. In *2016 45th International conference on parallel processing (ICPP)*, pages 67–76. IEEE, 2016.

[23] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

[24] Z. Nazir. Cnn cuda libraries performance. `https://github.com/zhumakhan/CNN_Cuda_libraries_performance/blob/master/dataset.csv`, 2022.

[25] Kien Nguyen, Clinton Fookes, Arun Ross, and Sridha Sridharan. Iris recognition with off-the-shelf cnn features: A deep learning perspective. *IEEE Access*, 6:18848–18855, 2017.

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

[27] NVIDIA Corporation. Nvidia profiler user's guide. `https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/`, Feb. 13, 2014.

[28] NVIDIA Corporation. Nvidia geforce 10 series graphics cards. `https://www.nvidia.com/en-eu/geforce/10-series/`, Nov. 15. 2015.

[29] NVIDIA Corporation. Nvidia tensorrt, Nov. 15, 2015.

[30] NVIDIA Corporation. Nsight graphics. `https://docs.nvidia.com/nsight-graphics/UserGuide/`, Nov. 1, 2022.

[31] NVIDIA Corporation. Nvidia profiler user's guide. `https://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference`, Dec. 8, 2022.

[32] Hiroyuki Ootomo and Rio Yokota. Reducing shared memory footprint to leverage high throughput on tensor cores and its flexible api extension library. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 1–8, 2023.

[33] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[34] Markus N Rabe and Charles Staats. Self-attention does not need $o(n2)$ memory. *arXiv preprint arXiv:2112.05682*, 2021.

[35] Joseph Redmon. Darknet: Open source neural networks in c. `http://pjreddie.com/darknet/`, 2013–2016.

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

[37] Siddharth Srivastava and Gaurav Sharma. Omnivec: Learning robust representations with cross modal sharing. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 1236–1248, 2024.

[38] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[39] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *International Conference on Machine Learning*, pages 10347–10357. PMLR, 2021.

[40] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[42] Micaela Verucchi, Gianluca Brilli, Davide Sapienza, Mattia Verasani, Marco Arena, Francesco Gatti, Alessandro Capotondi, Roberto Cavicchioli, Marko Bertogna, and Marco Solieri. A systematic assessment of embedded neural networks for object detection. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 937–944. IEEE, 2020.

[43] Leyuan Wang, Zhi Chen, Yizhi Liu, Yao Wang, Lianmin Zheng, Mu Li, and Yida Wang. A unified optimization approach for cnn model inference on integrated gpus. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[44] S. Winograd. Arithmetic complexity of computations. *Siam, volume 33*, 1980.

[45] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of cnn and rnn for natural language processing. *arXiv preprint arXiv:1702.01923*, 2017.

[46] Jiahui Yu, Zirui Wang, Vijay Vasudevan, Legg Yeung, Mojtaba Seyedhosseini, and Yonghui Wu. Coca: Contrastive captioners are image-text foundation models. *arXiv preprint arXiv:2205.01917*, 2022.

[47] Yujia Zhai, Chengquan Jiang, Leyuan Wang, Xiaoying Jia, Shang Zhang, Zizhong Chen, Xin Liu, and Yibo Zhu. Bytetransformer: A high-performance transformer boosted for variable-length inputs. In *2023 IEEE International*

Parallel and Distributed Processing Symposium (IPDPS), pages 344–355. IEEE, 2023.

[48] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.