Towards an Accurate Latency Model for Convolutional Neural Network Layers on GPUs

Jinyang Li*, Runyu Ma†, Vikram Sharma Mailthody*, Colin Samplawski‡,
Benjamin Marlin‡, Songqing Chen†, Shuochao Yao†, Tarek Abdelzaher*
* University of Illinois, Urbana-Champaign,
{jinyang7, vsm2, zaher}@illinois.edu
† George Mason University,
{rma5, shuochao, sqchen}@gmu.edu
‡ University of Massachusetts Amherst,
{csamplawski, marlin}@cs.umass.edu

MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM) | 978-1-6654-3956-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/MILCOM52596.2021.9652907

***Abstract*—Convolutional Neural Networks (CNN) have shown great success in many sensing and recognition applications. However, the excessive resource demand remains a major barrier against their deployment on low-end devices. Optimizations, such as model compression, are thus a need for practical deployment. To fully exploit existing system resources, platform-aware optimizations emerged in recent years, where an execution-time model becomes a necessity. However, non-monotonicity over the network configuration space makes execution time modeling a challenging task. Data-driven approaches have the advantage of being portable over different platforms by treating the hardware and software stack as a black box but at the cost of extremely long profiling time. On the other hand, analytical models can be found in the architecture and system literature that do not need heavy profiling but require laborious analysis by domain experts. In this paper, we focus on building a general latency model for convolutional layers that account for the majority of the total execution time in CNN models. We identify two major non-linear modes in the relationship between latency and convolution parameters, and analyze the mechanism behind them. The resulting model has better interpretability and can reduce profiling workload. The evaluation results show that our model outperforms baselines on different platforms and CNN models.**

I. Introduction

Convolutional neural networks (CNN) have shown great success on various sensing and recognition applications. Due to their high computational intensity, optimizations such as model compression [1] are often required in deployment, especially on edge devices and mobile platforms. These platform-aware optimizations exploit both the redundancy in the model and characteristics of the underlying hardware and software stack. Unfortunately, due to the very large parameter space of deep learning models, it is infeasible to search for the optimum by brute force. Therefore, a performance model that describes the relationship between the network configuration and the corresponding execution latency is needed.

In addition to the huge search space, a key factor that makes latency prediction very challenging is the non-monotonic and nonlinear relationship between latency and input scale [2, 3]. Yao et al. [2] reported that the convolution execution time on a mobile CPU showed a jagged fluctuation as the number of input and output channels increased. Zhang et al. [3] reported that this phenomenon also exists on mobile GPUs and VPUs, but with different modes. Existing latency models often ignore such phenomena or use learning-based methods [2–7] to avoid investigating the mechanisms that cause such irregularities. Although some models have high accuracy, they usually need a significant amount of profiling to collect enough training data. Even models based on learning methods with strong interpretability, such as decision trees [2, 3], often cannot provide real insights to explain these irregularities. On the other hand, analytical models [8–10] can be found in the architecture and system community that have fewer parameters and thus only need lightweight benchmarks to estimate. These analytical models require in-depth analysis (e.g., cache miss rate analysis, bottleneck analysis) of the hardware by domain experts and they often consider only low-level software and hardware factors, instead of covering the entire software and hardware stack like the learning-based models.

In this work, we try to bridge the gap between the analytical methods and learning-based methods. By comparing the profiling results with various non-linear effects caused by hardware and software factors, we find two major nonlinear modes in convolutional layer execution latency (Fig. 1). The step-like nonlinear mode is attributed to a hardware quantization effect caused by block-level parallelism in the GPU. The other non-monotonic change is caused by algorithm auto-tuning that exists at multiple levels in the software stack. We then propose a novel convolutional layer latency prediction model that, given an input convolution configuration, predicts which convolution kernel implementation will be used and uses the corresponding latency profile to predict the execution time.

The main contributions of this work are: (1) we identify two major nonlinear modes in the convolutional layer execution latency on GPUs and analyze the mechanisms behind them. (2) Based on the analysis, we propose a latency prediction model for convolutional layers that consists of a kernel latency model and a kernel selection model. The kernel latency model utilizes the fact that the computation performed by a GPU kernel is divided into fixed-size tiles. By estimating the size of the tile, the latency model can predict where step-like changes will occur thereby reducing profiling overhead (by avoiding

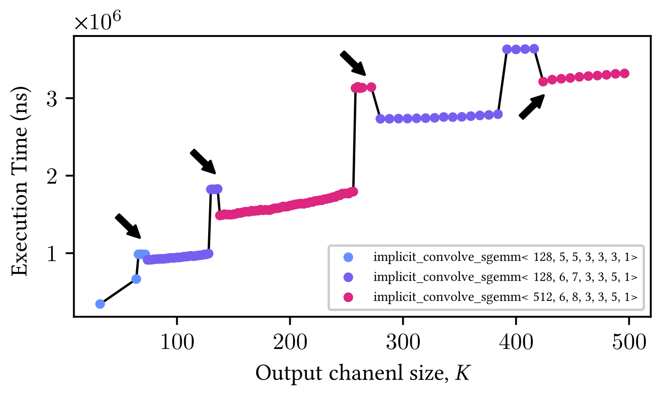


Fig. 1: Nonlinear modes in the convolutional layer execution time. The execution times and the cuDNN kernels used to compute one convolutional layer on P1 (Section IV) are recorded as we change the number of the output channels. The curve overall shows a step-like nonlinear pattern but there is also a non-monotonic pattern (marked by black arrows).

profiling the plateaus). The kernel selection model learns the heuristics used by the algorithm auto-tuning from the profiling dataset to predict where kernel switch will happen. We evaluate the proposed model on two platforms and two CNN models. The results show that the proposed method can achieve 91.29% and 59.6% accuracy on two experiment platforms, respectively (defined as the percent of predictions that fell within $\pm 10\%$ from ground truth). The accuracy of the nearest competitor was less than 33%.

II. Background and Motivation

GPU Architecture and Programming Model. The graphics processing unit (GPU) consists of many *Streaming Multiprocessor (SMs)* [11, 12] connected to a large last level cache using a crossbar interconnect and a high bandwidth global memory. Each SM consists of multiple sub-cores or threads, tens of kilobytes of register space, tens of kilobytes of a unified L1 cache and scratchpad. Modern GPU SMs [13, 14] consist of many functional units such as arithmetic operators, load/store engines and more recently, dedicated hardware accelerators for matrix multiplication (termed as TensorCores in NVIDIA GPUs).

A GPU programmer who uses CUDA can launch a compute kernel on the GPU with thousands to millions of threads organized in a grid and thread-blocks [11]. A grid is nothing but a collection of thread blocks organized to exploit parallelism within and across SMs. Each thread-block is mapped to an SM based on the resources available at run-time. When scheduling work within the SM, each SM further divides the thread-block into *warps*, groups of 32 threads. Modern GPU SMs can have up to four parallel warp schedulers to execute the work described in the compute kernel. Threads in a thread-block can synchronize and share data within the SM using shared memory. The dimensions and sizes of the gird and the thread block are specified by the caller of the kernel.

GEMM-based Convolution. The forward convolutional layer computes the convolution of an input tensor $\mathsf{X} \in \mathbb{R}^{N\times C\times H\times W}$ and a filter tensor $\mathsf{F} \in \mathbb{R}^{K\times C\times R\times S}$, producing an output tensor $\mathsf{Y} \in \mathbb{R}^{N\times K\times P\times Q}$. The computation is defined by the following equation

$$\mathsf{Y}_{n,k,p,q} = \sum_{c=0}^{C-1}\sum_{r=0}^{R-1}\sum_{s=0}^{S-1} \mathsf{X}_{n,c,p+r,q+r} \times \mathsf{F}_{k,c,r,s}$$

where N is the batch size, C is the number of input channels, H is the height of the input image, W is the width of the input image, K is the number of output channels, P is the height of the output image, Q is the width of the output image, R is the height of the filter and S is the width of the filter.

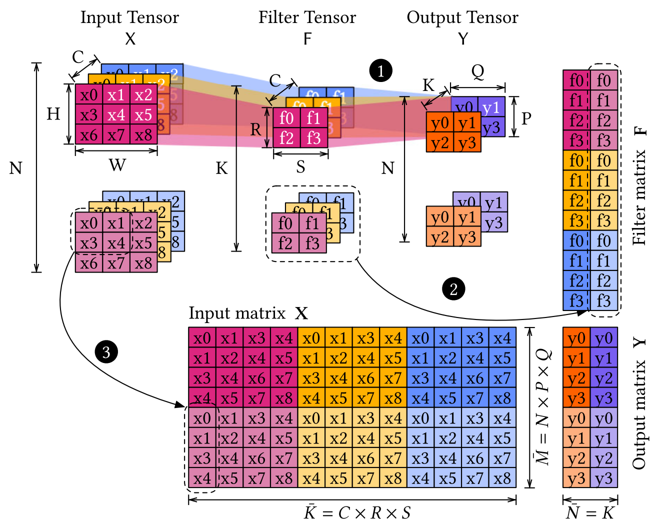


Fig. 2: A GEMM-based convolution example.

An example [8] is illustrated in Fig. 2. The element $\mathsf{Y}_{0,0,0,1}$ in the output tensor (the white y_1 in the purple channel) is computed by elementwise multiplying the 3-D slice $\mathsf{X}_{0,:,1:2,1:2}$ of the input tensor (highlighted in white) with the 3-D slice $\mathsf{F}_{0,:,:,:}$ of the filter tensor (highlighted in white), then summing up the the result and offsetting by the bias ❶.

The convolution kernel implementation on GPUs can be performed in several ways: (a) shared-memory convolution where the filter matrix F is stored in the shared memory, (b) using wino-grad convolution algorithm and (c) with the help of general matrix-multiplication (GEMM) implementation [15]. Among these implementations, GEMM-based convolution implementation is the most efficient algorithm [15–17]. In this implementation. the filter tensor F is reshaped into a matrix $\mathbf{F} \in \mathbb{R}^{K\times CRS}$ ❷. The data in the input tensor X is unfolded and replicated to shape into a matrix $\mathbf{X} \in \mathbb{R}^{CRS\times NPQ}$ according to the access pattern of the convolution ❸. The computation now becomes a single general matrix multiplication (GEMM) which can be computed with highly optimized linear algebra routines. But lowering the input tensor into a matrix requires data duplication increasing memory overhead and requires two CUDA kernel launches which can become prohibitive. To address this, *implicit-GEMM-based convolution* performs conceptually the same computation but avoids constructing the matrices explicitly in the global memory by carefully manipulating pointers and predicates during computation [15, 18].

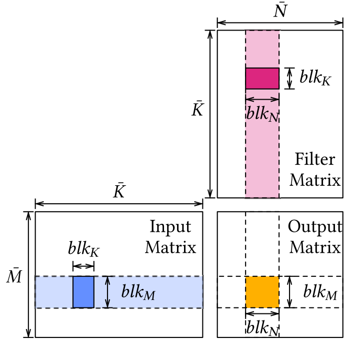


Fig. 3: Tiled outer product GEMM [17]

As shown in Fig. 3, efficient GEMM implementation blocks and tiles the matrices to fully exploit the concurrency in hardware and improve memory locality. Each output tile (yellow block of size $blk_M \times blk_N$) is computed by iterating the tiles along the $\bar{K}$ dimension in the input and output matrix and accumulating the results; at each iteration, the outer products of the columns in the input tile (blue block of size $blk_M \times blk_K$) and the rows in the filter tile (red block of size $blk_K \times blk_N$) are computed and accumulated. The computation of each output tile is performed by a thread block and, therefore, the computation of all tiles, i.e., the iteration along dimension $\bar{M}$ and dimension $\bar{N}$, can be performed in parallel. The tiling technique can be further applied at warp-level and thread-level to fully exploit parallelism and data locality, forming a hierarchical GEMM structure [18].

III. System Design and Implementation

A. Kernel Latency Model

We run benchmarks and analyze various software and hardware factors that can lead to non-linearity. The step-like mode can best be described by a quantization effect caused by the GPU architecture and execution model. Fig. 4 shows the execution time and grid size (number of thread blocks launched) of each kernel in Fig. 1, respectively. Execution times are visualized as scatters and the black curve shows how the gird size changes.

The *tile quantization effect* refers to the phenomenon of "discontinuous" changes in execution time when the input size is changed "continuously" (minimum step size) because of the fixed tile size [19]. An example of matrix multiplication computed with kernel of size $blk_M \times blk_N$ ($blk_M > 1$ and $blk_N > 1$) is illustrated in Fig. 5. The image on the left shows the ideal situation where the dimensions of the output matrix, $\bar{M} = 2blk_M$ and $\bar{N} = 2blk_N$, are divisible by the tile dimensions. In this case, the computation requires four fully-utilized thread blocks. However, if we increase the number of columns of the matrix by one, instead of allocating just enough resources to compute the additional $\bar{M} \times 1$ elements, two more thread blocks are created. Because of the SIMT execution model, the two additional thread blocks marked in red take the same amount of time to finish as the fully-loaded ones, even though most of the threads are not doing meaningful work. As a result, increasing one dimension of the input by one leads to a $50\,\%$ increase in execution time.

We can see clearly that the change in execution time in Fig. 5 matches the change in grid size, which is consistent with the tile quantization effects. These observations imply that we can estimate the execution time by calculating the grid size required for a given workload.

Let $\mathbf{s}_l$ denote the configuration of layer l of a L-layer CNN model. $\mathcal{S} = \{\mathbf{s}_l \mid l \in [1, L]\}$ represents the configuration of the whole network and $\mathcal{S}_{\text{conv}}$ represents the set of configurations of the convolutional layers. The configuration of a convolutional layer $\mathbf{s}_l \in \mathcal{S}_{\text{conv}}$ is a tuple of parameters that define the convolution $\mathsf{s}_l = \langle N, C, H, W, K, R, S, P, Q \rangle$ where the convolution parameters have the same meaning as shown in Fig. 2. For brevity, the subscript of each parameter is omitted.

As introduced in Section II, the convolution described by $\mathbf{s}_l$ is lowered into a GEMM of an input matrix $\mathbf{X} \in \mathbb{R}^{\bar{M} \times \bar{K}}$ and a filter matrix $\mathbf{F} \in \mathbb{R}^{\bar{K} \times \bar{N}}$ where $\bar{M} = NPQ$, $\bar{N} = K$ and $\bar{K} = CRS$.

The tiling parameters blk_M, blk_N of the kernel i describe the tile size along $\bar{M}$ and $\bar{N}$. Tiling parameters of different kernels can take different values. For brevity, we omit the subscript i in the tile size parameters. The gird size g to compute the given convolution workload can be calculated as follows:

$$g = \lceil \frac{\bar{M}}{blk_M} \rceil \lceil \frac{\bar{N}}{blk_N} \rceil \tag{1}$$

Then the execution time of the kernel is estimated as follows:

$$t = g\bar{K}t_K + m_{\text{out}}t_{\text{out}} + m_{\text{in}}t_{\text{in}} + \gamma \tag{2}$$

t_K is the average execution time of each tile; $m_{\text{out}}t_{\text{out}}$ and $m_{\text{in}}t_{\text{in}}$ are the memory operation overhead of input and output, respectively; the amount of memory operations is estimated as $m_{\text{in}} = NCHW + KCRS$ and $m_{\text{out}} = NPQK$; γ is a constant.

The results of the profiling tasks form a dataset denoted by $\mathcal{D}$

$$\mathcal{D} = \{\mathbf{d}_j \mid \langle \mathbf{s}_j, g'_j, t'_j, w'_j \rangle,\ j \in [1, J]\}$$

where $\mathbf{s}_j$ is the convolution configuration of the task j; g'_j is the measured grid size; and t'_j is the measured execution time. The tiling factors can be obtained by solving the following optimization problem:

$$\tau = \arg\min_{\tau} \sum_{j=1}^{J} |g'_j - g_j| \tag{3}$$

where $\tau = \langle blk_M, blk_N \rangle$ and g_j is the grid size calculated with τ using Eq. (1). Finally, the remaining parameters are calculated as

$$\phi = \arg\min_{\phi} \sum_{j=1}^{J} |t'_j - t_j| \tag{4}$$

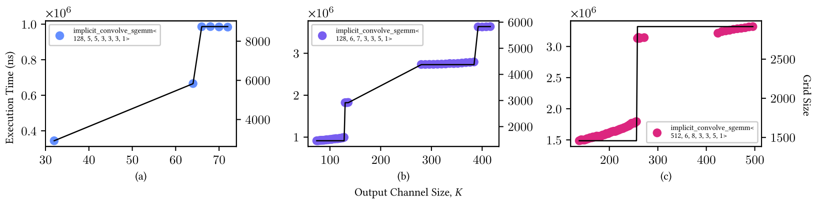


Fig. 4: Grid size

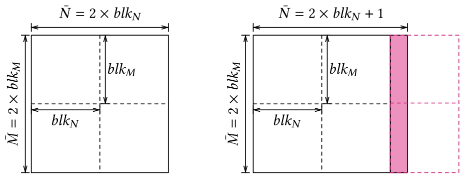

$\bar{N} = 2 \times blk_N$
$\bar{M} = 2 \times blk_M$
blk_M
blk_N
$\bar{N} = 2 \times blk_N + 1$
$\bar{M} = 2 \times blk_M$
blk_M
blk_N

Fig. 5: Tile Quantization Effect

where $\phi = \langle t_K, t_{\text{out}}, t_{\text{in}}, \gamma \rangle$ and t_j is the execution time calculated with ϕ using Eq. (2).

B. Kernel Selection Model

Let $\theta_i = \langle \tau, \phi \rangle$ denote the parameters of the Kernel Latency Model of kernel i. We define $t(\mathbf{s}_j; \theta_i)$ to be the function that, given a convolution configuration $\mathbf{s}_j$ as the input, computes the execution time estimate using θ_i and Eq. (1) to Eq. (2). What we need next is an overall model, a function $t(\mathbf{s}_j)$ that gives the estimate only based on the input.

A natural and straightforward attempt to define $t(\mathbf{s}_j)$ may look like the following:

$$i' = \arg\min_i t(\mathbf{s}_j | \phi_i) \tag{5}$$

$$t(\mathbf{s}_j) = t(\mathbf{s}_j; \theta_{i'}) \ . \tag{6}$$

However, to optimize performance, most deep learning frameworks and low-level math libraries will try to find the most performant algorithm for the computational workload. This process is often abstracted away from the user; hence the name "algorithm auto-tuning".

There are two ways to auto-tune the algorithm. The first one is profiling-based auto-tuning, which executes the workload with algorithms/kernels available and chooses the fastest one. It is accurate but slow since the computation is executed for multiple times. Alternatively, the runtime can use predefined heuristics to determine which algorithm/kernel should be chosen. It is fast because the workload is not really computed, and accurate in many cases, but the decisions are not always optimal [20]. Unfortunately, the heuristics used by cuDNN are not publicly available.

To overcome the uncertainty in the heuristics and the visibility issue, we use a learning-based method for the kernel selection modeling. The features used to train the model are (1) convolution parameters $\mathbf{s}_l$, (2) GEMM dimensions $\bar{M}$, $\bar{N}$ and $\bar{K}$, (3) Kernel Latency Model features g, m_{in} and m_{out}, and (4) latency predictions of each kernel.

IV. Evaluation

We evaluate the proposed method on ResNet50 V2 and Inception V3 on two platforms: **P1** is a desktop server equipped with a GeForce RTX 2080 Ti GPU and running Ubuntu 18.04 (Linux Kernel 5.4.0-73-generic-x86_64), CUDA 10.2.89, NVIDIA Driver 460.80 and cuDNN 7.6.5. **P2** is NVIDIA Jetson TX2 equipped with a NVIDIA Tegra X2, running Ubuntu 18.04 (Linu kernel 4.9.201-tegra-aarch64), CUDA 10.2.89, NVIDIA Driver L4T 32.5.1, and cuDNN 8.0.0.

The proposed method is compared with two baselines. The first is a FLOPs-based estimator obtained by fitting a linear model of total number of multiply and add operations using the profiling data. The second is FastDeepIoT. The metrics used are Root Mean Square Error (RMSE) and $\pm 10\%$ accuracy, defined as the percentage of the predictions that fall within the range of ground truth $\pm 10\%$.

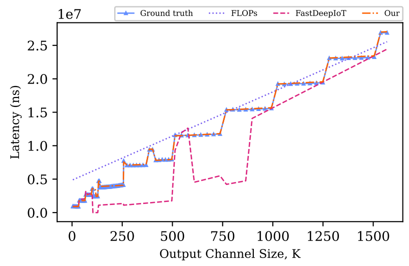
We generate test cases from the convolution configurations in each neural network model by finding the unique combinations of H, W, R, S, stride size, dilation size and padding size. For each test case we generate profiling tasks with different number of input channels and output channels. We profiled and collected results from 10,945 ResNet50 V2 tasks and 5,912 Inception V3 tasks. Due to the step-like pattern, we only need to sample near the rising edge of the step, which greatly reduces the profiling workload.

The predictions accuracy and RMSE of the proposed method and baselines on ResNet50 V2 and Inception V3 on platform 1 and 2 are reported in Table I. On average, our method achieved 91.28% and 59.60% accuracy on each platform respectively. In comparison, the $\pm 10\%$ accuracy of FastDeepIoT is only 22.25% and 15%; the $\pm 10\%$ accuracy of FLOPs-based baseline is just 2.56% and 7.57%. The results of ResNet50 V2 on P1 and Inception V3 on P1 show that our method can adapt to CNN models with various convolution configurations.

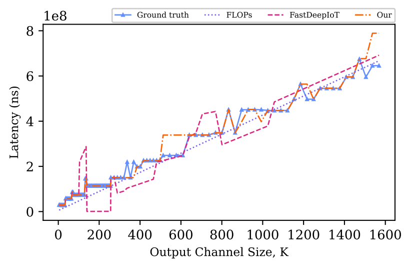
Fig. 6 shows the detailed result of Case 0 of ResNet50 V2 on P1. The performance of each method under different input channels is plotted separately. Our method utilizes the knowledge about the quantization effect and thus can fit the step-like curve well with a small amount of profiling data. On the other hand, FastDeepIoT uses a binary tree structure to model the latency where each leaf is a linear regression model. It works well on mobile CPUs, where the non-linearity is mainly zig-zags caused by memory-related factors. However, on a GPU, it struggles to fit the step-like curve with its tree structure model. Another reason that FastDeepIoT does not work well is that the profiling dataset is not large enough. To prevent overfitting, FastDeepIoT stops growing the tree when the leaf has too few data points or the fitting error is below 5%. When the dataset is small, the tree stops growing more often due to the lack of data, instead of meeting the accuracy condition. There are a few cases where our model has low accuracy. We find that in those cases the convolution filter

TABLE I: Prediction accuracy and root mean square error.

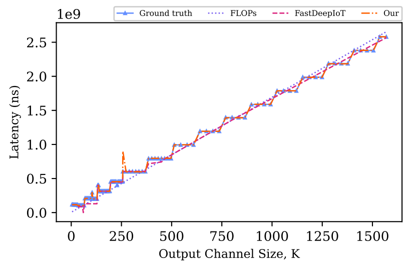
	Our		FLOPs		FastDeepIoT	
	±10% Accuracy	RMSE (ms)	±10% Accuracy	RMSE (ms)	±10% Accuracy	RMSE (ms)
ResNet50 V2 on P1	91.34%	0.10	2.25%	4.70	11.90%	1.90
ResNet50 V2 on P2	59.60%	6.70	7.57%	48.35	15.00%	42.58
Inception V3 on P1	91.22%	0.03	2.87%	2.31	32.60%	0.40



(a) $C = 8$



(b) $C = 256$



(c) $C = 1024$

Fig. 6: The predictions and ground truth of three different C values of ResNet50 V2 on RTX 2080 Ti. The proposed method out performs the baselines in all situations. (a) FLOPs-based method only captures the general trend. FastDeepIoT struggles to fit the step-like curve. (b) When the training data is not sufficiently large, FastDeepIoT may produce false non-linearity which will misguide the compressor in model compression. (c) In certain case, FastDeepIoT degenerates into FLOPs baseline.

size is 1×1 and the image size is very small (e.g., 10×10) thus computed as a normal matrix multiplication, instead of convolution.

Our method achieved 59.60% accuracy on P2. On platform P2, the measured latency curve is more "noisy". The GPU on P2, NVIDIA Tegra X2, is an embedded/mobile GPU, much weaker than the RTX 2080 Ti on P1. Therefore, it is easier to encounter resource bottlenecks such as bandwidth and memory on this platform, resulting in non-linearities not modeled by our method.

The results of Case 0 in all three experiments have high accuracy as well as high RMSE. The reason is that overall our method gives predictions close to ground truth but diverged from the ground truth when $K \geq 600$. However, it still correctly predicted the positions of the raising edge of the step. It means that the height of the step t_K is underestimated. Case 0 is the first convolutional layer in ResNet50 where the filter size is 7×7 and image size is 230×230. Its computation intensity is significantly higher than that of the later layers where both image size and filter size are small. This type of layers usually appear only once in the network and can be fitted separately.

V. Related Work

There are analytical performance models proposed by previous work for GEMM-based algorithms [8, 9] or general workload on GPU [10]. It usually requires a great amount of details of the hardware and in-depth analysis of memory traffic pattern and various computation bottlenecks which is difficult to port to various platforms. These models also do not consider factors that can cause non-linearity existed in the upper level of the software stack, such as algorithm auto-tuning.

Much existing work uses learning-based models to predict neural network execution latency. There are methods that use FLOPs as the main feature to predict latency [4, 5]. Although simple and intuitive, they are not accurate. More sophisticated methods also consider other features such as IO overhead and use linear model [6], decision tree based models [2, 3], or neural networks [7] to predict latency. However, none of these methods considers and analyzes the mechanisms that cause non-linearities. Our work is distinguished from these approaches in that we attribute a dominant non-linearity to the hardware quantization effect and algorithm auto-tuning and utilize this information to reduce profiling workload.

VI. Conclusion and Future Work

In this paper, we proposed a novel hybrid latency model for convolutional neural networks. Different from existing pure data-driven approaches, we attribute the dominant non-linearity in convolutional layer execution latency found on GPUs to the hardware quantization effect caused by the GPU's execution model and algorithm auto-tuning in the software stack. With this knowledge, the proposed method reduces the profiling workload and provides better model interpretability.

The experimental results show that our method outperforms the baselines significantly across the platforms and across different neural network models used in the evaluation. The average (±10%) accuracy that the proposed method achieved is 91.28% on a desktop/server GPU and 59.60% on an embedded GPU, which is 4.01× and 3.97× better than the baselines, respectively. The proposed method is able to model the non-linearities caused by hardware quantization effects and algorithm auto-tuning. In the case of an extreme computational workload, the prediction accuracy may be further improved by dynamically adjusting model parameters. The nonlinear modes observed and the proposed performance model should also be applicable to other deep learning models that use convolutional layers. Evaluation on other types of neural network models, such as NLP models, and the impact of reducing latency on battery life are left for future work.

Acknowledgements

Research reported in this paper was sponsored in part by the Army Research Laboratory under Cooperative Agreement W911NF-17-2-0196 and NSF under award CPS 20-38817, and in part by The Boeing Company. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory, NSF, Boeing, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] S. Yao, Y. Zhao, A. Zhang, L. Su, and T. Abdelzaher, "DeepIoT: Compressing Deep Neural Network Structures for Sensing Systems with a Compressor-Critic Framework," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '17. New York, NY, USA: ACM, 2017, pp. 4:1–4:14.

[2] S. Yao, Y. Zhao, H. Shao, S. Liu, D. Liu, L. Su, and T. F. Abdelzaher, "FastDeepIoT: Towards understanding and optimizing neural network execution time on mobile and embedded devices," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems, SenSys 2018, November 4-7, 2018*. Shenzhen, China: ACM, 2018, pp. 278–291.

[3] L. L. Zhang, S. Han, J. Wei, N. Zheng, T. Cao, Y. Yang, and Y. Liu, "Nn-Meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. Virtual Event Wisconsin: ACM, Jun. 2021, pp. 81–93.

[4] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "AMC: AutoML for Model Compression and Acceleration on Mobile Devices," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 784–800.

[5] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning Filters for Efficient ConvNets," *arXiv:1608.08710 [cs]*, Mar. 2017.

[6] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A Performance Model for Deep Neural Networks," Nov. 2016.

[7] Ł. Dudziak, T. Chau, M. S. Abdelfattah, R. Lee, H. Kim, and N. D. Lane, "BRP-NAS: Prediction-based NAS using GCNs," *arXiv:2007.08668 [cs, eess, stat]*, Jan. 2021.

[8] S. Lym, D. Lee, M. O'Connor, N. Chatterjee, and M. Erez, "DeLTA: GPU performance model for deep learning applications with in-depth memory system traffic analysis," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019, March 24-26, 2019*. Madison, WI, USA: IEEE, 2019, pp. 293–303.

[9] J. Lai and A. 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 (CGO)*, Feb. 2013, pp. 1–10.

[10] K. Zhou, G. Tan, X. Zhang, C. Wang, and N. Sun, "A performance analysis framework for exploiting GPU microarchitectural capability," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 1–10.

[11] NVIDIA Corporation, "CUDA C++ Programming Guide," May 2021. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[12] J. Burgess, "RTX on—The NVIDIA Turing GPU," *IEEE Micro*, vol. 40, no. 2, pp. 36–44, Mar. 2020.

[13] "Nvidia a100 gpu architecture whitepaper," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[14] "Nvidia tesla v100 gpu architecture whitepaper," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[15] D. B. Kirk and W. mei W. Hwu, in *Programming Massively Parallel Processors (Third Edition)*, third edition ed. Morgan Kaufmann, 2017.

[16] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," *arXiv:1410.0759 [cs]*, Dec. 2014.

[17] N. Corporation, "Deep Learning Performance Documentation," Jul. 2020. [Online]. Available: http://docs.nvidia.com/deeplearning/frameworks/dl-performance-matrix-multiplication/index.html

[18] NVIDIA Corporation, "CUTLASS: CUDA Templates for Linear Algebra Subroutines," Feburary, 2021. [Online]. Available: https://github.com/NVIDIA/cutlass

[19] ——, "Matrix Multiplication Background User Guide," Jul. 2020. [Online]. Available: http://docs.nvidia.com/deeplearning/frameworks/dl-performance-matrix-multiplication/index.html

[20] C. Li, A. Dakkak, J. Xiong, and W.-m. Hwu, "Benanza: Automatic μBenchmark Generation to Compute "Lower-bound" Latency and Inform Optimizations of Deep Learning Models on GPUs," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. New Orleans, LA, USA: IEEE, May 2020, pp. 440–450.