Fast and Accurate DNN Performance Estimation across Diverse Hardware Platforms

Vishwas Vasudeva Kakrannaya
Pennsylvania State University
University Park, PA, USA
vvv5127@psu.edu

Siddhartha Balakrishna Rai
Pennsylvania State University
University Park, PA, USA
sub1089@psu.edu

Anand Sivasubramaniam
Pennsylvania State University
University Park, PA, USA
anand@cse.psu.edu

Timothy Zhu
Pennsylvania State University
University Park, PA, USA
timothyz@cse.psu.edu

***Abstract*—Performance modeling is an important tool for many purposes such as designing hardware accelerators, improving scheduling, optimizing system parameters, procuring new hardware, etc. This paper provides a new methodology for constructing performance models for Deep Neural Networks (DNNs), a popular machine learning workload. Prior works require running DNNs on existing hardware, which may not be available, or simulating the computation on futuristic hardware, which is slow and not scalable. We instead take an analytical approach based on analyzing the raw operations within DNN algorithms, which allows us to estimate performance across any hardware, even hardware that is in the process of being designed. Evaluations show our approach is fast and gives a good first order approximation ($\pm 10 - 15\%$ accuracy) across many DNNs and hardware platforms including GPUs, CPUs, and a futuristic Processing In Memory (PIM) accelerator called BLIMP.**

***Index Terms*—Analytical models, DNN models, Heterogeneous systems, Performance modeling**

I. Introduction

Deep Neural Networks (DNN) are commonplace for numerous machine learning tasks. This has led to a flurry of activity in the past decade from several fronts - developing new models for applications, fine tuning parameters for existing models, mapping these models on existing hardware for maximum efficiency, designing and developing new hardware for running these models, deploying and operating datacenter infrastructure to host these models in a cost-efficient manner, etc. At the heart of all these efforts are answers to the following questions: *how would model X perform on hardware Y (whether the hardware is already present, or is to be procured, or is being hypothesized)? Are we using the best algorithms to run a given model on the hardware? What would be the ideal configuration parameters (batch size, number of Processing Engines, etc.) for running these models on the given hardware?* For the first time, this paper presents a simple model that can give a quick first order approximation of the performance of a given DNN on a given hardware platform to answer these questions.

There are numerous model parameters, algorithmic choices, and hardware platforms (both existing and emerging), that are continuously evolving to run large DNNs. On the algorithmic front, designers are constantly improving their models, introducing new layers to optimize the model for real time constraints that is often the case for ML inference. Understanding how their models would perform on existing hardware is at the crux of their designs.

On the hardware front, computer architects are continuously innovating their hardware with new features/accelerators (e.g., TPUs, PIM accelerators) to improve model performance. While industry and researchers often test new designs using cycle accurate simulators, running large DNN models at scale is virtually impossible on such simulators. A first order model that can quickly predict performance would help architects understand "what-if" trade-offs in their designs and hypothesize how they would perform even if the hardware does not exist today.

Programmers and runtime systems need to map DNN models efficiently onto the given hardware. However, this is a non-trivial problem - even if there are numerous libraries to perform common tasks (e.g., matrix multiply). Each hardware has its own nuances, impacting how each algorithm can perform despite the same high level functionality. Further, the efficiency of the mapping can be impacted by the parameters (e.g., tensor dimensions/lengths can impact cache hit rates) that they need to operate on. Programmers can clearly benefit from a framework to quickly estimate what algorithms should be employed for the various parameters and high level operations.

Finally, infrastructure providers (e.g., cloud/datacenters) often have a diversity of existing hardware and options/budget to procure new systems to host ML workloads. Such hardware is expensive, with a high demand for their capabilities in this AI era. Providers not only have to grapple with how best to allocate their budget across different options with different price-performance trade-offs, but also dynamically decide how to allocate workloads onto their existing diverse/heterogeneous hardware resources to meet overall SLOs.

Despite such widespread needs, there is a dearth of tools to quickly estimate performance of these DNN models on diverse hardware. At one end, simulators are flexible in modeling hardware, even hardware that may not exist, but are time-consuming to build and more time-consuming (or intractable) to execute for large DNNs. At the other end, one can empirically benchmark models on different hardware, but it may not accurately reflect the performance on new hardware or even slight changes to existing hardware (e.g., larger caches, fast interconnects, etc.).

To address these needs, this paper presents a simple an-

alytical model to quickly give a first order approximation (even a $\pm 10 - 15\%$ accuracy suffices for many of the above identified purposes) of DNN execution time on any hardware, current or hypothetical. The goal of our model is to capture the essentials of the DNN computation and the hardware, without over-burdening the user, while still accurately capturing the nuances of the interactions between the hardware and software. Towards this end-goal, this paper makes the following contributions:

1) We find that three important software characteristics - the number of floating-point operations, the data transfer costs, and the effective parallelism - are sufficient for estimating DNN execution time. From the hardware perspective, simple characteristics such as number of Processing Engines (PEs), Processing Speed, and memory bandwidth, which can be obtained from data sheets and/or microbenchmarks, are sufficient for estimating performance. We find that other factors have a smaller impact on accuracy, and we explain how they can be addressed at the expense of a more complex model.
2) Based on these easily accessible parameters, we develop a methodology for creating simple analytical models that can reasonably estimate the execution time of individual kernels and layers of DNN models based on the algorithm used. We combine these estimates to provide an overall estimate for the entire DNN for various hardware platforms.
3) We evaluate our analytical model across many DNNs and hardware platforms, including an emerging hardware platform (BLIMP). Our evaluations show the accuracy of our analytical model across CPUs and GPUs (with and without tensor cores) for various DNNs.
4) We demonstrate use-cases for our model in (i) debugging performance problems (e.g., finding inefficient kernels) and (ii) predicting the execution time for a futuristic BLIMP-PIM accelerator.

II. Related Work

Several works have investigated performance prediction of DNN models across a variety of hardware. The approaches can be broadly placed into four categories: Empirical approaches, Simulation based approaches, Machine learning based approaches, and Analytical model based approaches.

Empirical approaches: Empirical approaches work by running the application on the given hardware, measuring the performance, and then extending the trend to other applications in the future. The main limitation is they require execution on the hardware platform to perform the analysis. Most of the works using an empirical approach focus on the training phase of the DNN models, although the insights from these works can be extended to inference as well. [1], [2] use an empirical approach to analyze the performance of the training phase using representative kernels on the cloud and other platforms. [3] does the performance analysis on a multi-GPU setup. Works like [4] provide benchmarks and analysis tools to predict the performance. Although they provide valuable insights regarding the kernels and predict the performance on a given hardware, *these approaches require access to the hardware and can not be extended to hypothetical hardware when designing new hardware.*

Simulation approaches: Simulation frameworks like Smaug [5] and other works like [6]–[8] are designed to accurately represent low level details. They are mainly used for estimating hardware resource efficiency and energy consumption. Although these provide great insights to the user, *they require intimate knowledge of the hardware and the software (libraries) along with lot of engineering efforts. They also take a significant amount of time to execute for accurate estimation.*

Machine learning (ML) based approaches: In the ML based approaches like [9]–[15], ML models are designed to predict the performance of DNN kernels across various hardware platforms. These have been used in scenarios such as scheduling DNN models, improving the utilization of hardware, etc. For example, [14] and [12] predict the performance of DNN inference using a transformer based model and Graph Neural network, respectively. CoDL [13] estimates the latency of DNN inference in heterogeneous platforms using a regression model that determines parameters for a detailed analytical model. DNNAbacus [9] focuses on predicting the performance and the memory usage of the DNN training process using a ML based model. [10] and [11] propose approaches that perform layer-by-layer analysis to provide overall estimation of the performance of DNN models. Although ML based models are convenient in predicting the performance accurately, they involve feature selection and a time consuming training process. Also, the hyper-parameters these models derive can often be restricted to the hardware on which they have been trained. *Hence it becomes difficult to extend the work across hardware platforms, especially for hypothetical hardware that do not exist.*

Analytical modeling approaches: Analytical models are the fastest approach to estimating performance. The most relevant work is Paleo [16], which focuses on performance prediction for the training phase of DNN models. Paleo considers the Big O notation of each operation (specific to the algorithm) and the workload size to determine the execution time for DNN training. Paleo focuses on GPUs including multi-GPU settings. In contrast, our work focuses on DNN inference, which operates at much smaller timescales (e.g., milliseconds vs hours). Additionally, our work is more general in supporting CPUs, GPUs, GPUs with tensor cores, and emerging hardware like BLIMP. Our design supports hypothetical hardware to answer what-if questions, whereas Paleo requires running benchmarks on every platform for every algorithm to obtain a PPP (Platform Percent of Peak) factor necessary for adjusting the estimation to be accurate. This makes the analytical model difficult to use unless one has access to the hardware under consideration.

In order to address these disadvantages, we propose a new approach based only on the specification sheet (or general FLOP rating and bandwidth measurements in cases where the specification sheet lacks this information). Our aim is to build a

fast and versatile analytical model which is *relatively agnostic of specific frameworks/libraries* while being general enough to estimate the execution time of the DNN model on any given hardware platform (commodity or prototypical/hypothetical ones). At the same time, the model needs to be simple enough to be easily specifiable by a relatively naive user, without requiring a wide spectrum of DNN models and/or hardware parameters that only an expert may be able to provide. *This way, our proposed analytical model can provide quick insights to both hardware and software designers for where to optimize the hardware and/or software without using empirically derived parameters.*

III. Modeling the performance of a DNN

The goal of the model is to explore what-if scenarios for the execution time of DNN models on various hardware platforms. In order to accommodate rapid changes occurring in both software and hardware fields and provide quick feedback for developers, our model should be simple enough to accept readily available parameters and general enough to accommodate a wide variety of hardware. The inputs to this model should be easily extractable either from the execution framework (like Pytorch or Tensorflow) or from the datasheets of the hardware. At the same time, accuracy cannot be compromised. To realize these goals, we need to identify the primary hardware and software factors that impact the performance of DNN models to be able to reasonably predict the execution time. To tackle this, we first need to understand the basic structure of DNN models.

Structure of the DNN models: A Deep Neural Network (DNN) model comprises a Data Flow Graph (DFG), where each node represents a DNN layer, and each layer is associated with a specific computation. Edges in the DFG represents the dataflow between the layers as shown in Figure 1. DNN layers perform operations on multidimensional arrays known as tensors. Depending on the operation (e.g., Convolution, ReLU, GeMM), each DNN layer transforms input tensors into output tensors. The output from one layer becomes the input for other layers based on the edges in the DFG of the DNN. The DFG is not necessarily a simple linear graph; it can include periodic divergences and convergences.

Various algorithms can implement operations in DNN layers. For instance, Convolution can be implemented using the Direct convolution algorithm, Winograd algorithm [17], FFT-based algorithm [17], [18], or implicit GeMM-based algorithm. Typically, DNN model developers specify operations (such as convolution) to frameworks like PyTorch [19] or TensorFlow [20]. Based on the dimensions of input tensors and the hardware platform, underlying libraries like Intel MKLDNN or CuDNN choose the optimal algorithm for the best performance.

The compute and the data movement costs: The execution of a DNN layer involves performing ALU operations on tensors obtained from memory. Therefore, the execution time of the DNN layer comprises both the time spent on computation and the time spent on data movement. Computation primarily involves applying fundamental operations like Multiply-and-ACCumulate (MACC) operations (for convolution) or Compare-and-Assign (for ReLU) inside nested loops to efficiently perform the specified layer operation. The compute time is thus a function of the number of these fundamental operations performed and the processing speed of the hardware.

Data movement on the other hand, entails transferring data from memory to the Processing Engine (PE). The data movement time is influenced by the number of bytes transferred between memory and the PE and the bandwidth of the interconnect. If there are multiple PEs, distributing computation among them can enhance performance. Moreover, multiple PEs can better utilize the interconnect between memory and PEs by saturating it. Equally, it can also create contention in the shared interconnect. Consequently, parallelism becomes another crucial factor in determining the execution time of a DNN layer. Therefore, by accurately calculating the compute time and data movement time, and by accounting for the parallelism, one can precisely estimate the execution time of a DNN layer.

Extending the layer-by-layer calculation to the entire DNN: DNN frameworks typically perform a topological sort and execute layers sequentially one after another. Since each layer is launched as kernels/functions separately (i.e., different layers are not executed in parallel), from the performance estimation point of view, the order in which DNN layers gets executed is irrelevant. That is, whether a specific layer gets executed as the first layer or tenth layer has no significance on the execution time of the DNN as long as the input and output tensor sizes are same. Hence the total execution time of the DNN model can simply be calculated by adding the execution time of individual layers.

In summary we use the following data as input to our model: From the DNN model, (i) the DataFlow Graph (DFG) of the neural network, (ii) the algorithm used in each of the layers (nodes) of this DFG, and (iii) the dimensions/sizes of the input/output tensors. From the hardware, (i) the number of processing elements (PEs), (ii) the processing speed (i.e. FLOP rating) of these PEs, and (iii) the bandwidth of the interconnect between the PE and the memory (typically DRAM).

Acquiring these inputs is relatively straightforward. On the DNN model side, input can be directly obtained from the framework. Similarly, input from the hardware side can be derived either from the specification sheet [21], [22] or by conducting micro-benchmarks [23], [24]. These inputs, although simple, are general enough to allow mapping of various hardware platforms such as CPU, GPU, TPU, and other accelerators into our modeling framework. In the following section, we elaborate on how these inputs are utilized to estimate the execution time of DNN models.

IV. Our analytical modeling methodology

Based on the ideas discussed in the previous section, our analytical modeling methodology can be summarized using

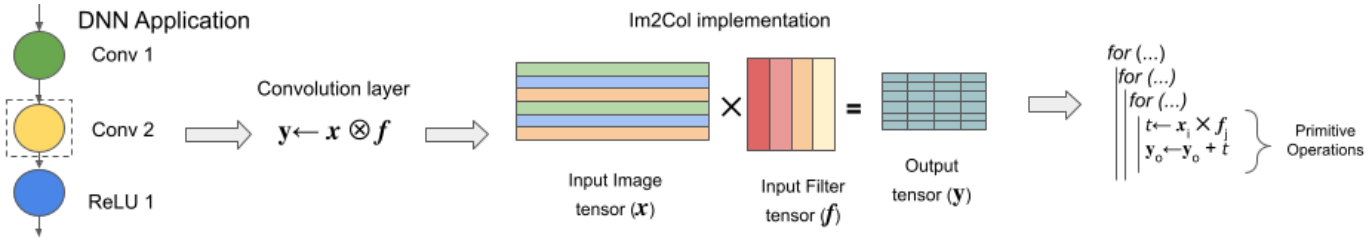


Fig. 1: Calculating the number of primitive operations in a DNN model.

the following formula for execution time:

$$\sum_{L \in layers} \left(\frac{fundamentalOps_L}{numPE_L * ProcSpeed} + \frac{dataMoved_L}{memoryBandwidth} \right)$$

Here $fundamentalOps_L$ represents the number of Floating point operations in layer L, which are based on the specific algorithm and input parameters/sizes. $dataMoved_L$ represents the data loaded and stored in layer L, which are based on the input/output sizes. $numPE_L$, $ProcSpeed$, and $memoryBandwidth$ represent the number of processing engines used in a layer, processing speed of a PE, and memory bandwidth of the hardware, respectively. Next, we provide intuition behind this expression and how each parameter in this expression is obtained.

A. Intuition behind the analytical model

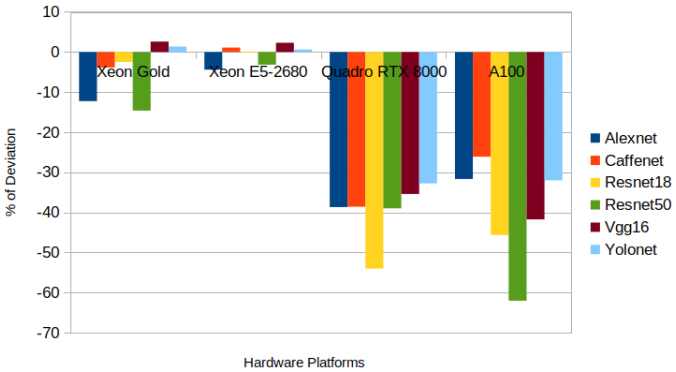


Fig. 2: Accuracy of analytical model for compute intensive convolution layers when only Compute time is considered.

The above mentioned expression has 2 components: (i) Compute time calculation and (ii) Data movement calculation. The intuition behind these have been explained in section III, and we add these 2 components to to calculate the execution time of a given layer. One may wonder, why addition is used instead of considering maximum of compute time and data movement time. When we considered just the compute time for compute intensive layers (convolution in this case), from Figure 2 we can see that, although the deviations are acceptable for CPUs, it is high for GPUs (positive deviation means over estimation and negative deviation means under estimation of execution time from our analytical model). But when both compute and data movement components are considered and added, the deviations arrived into reasonable level of accuracy which can be seen in section VI. *Hence both compute time and data movement times are added to get the execution time of a layer.*

#	Layer name	Input dimension $[n \times c \times h' \times w']$	Kernel size $r \times s$	Padding p	Strides s	Output Dimension $[n \times k \times h \times w]$
1	Conv2d-1	[1x3x224x224]	11x11	2	4	[1x64x55x55]
2	ReLU-2	[1x64x55x55]	-	-	-	[1x64x55x55]
3	MaxPool2d-3	[1x64x55x55]	3x3	0	2	[1x64x27x27]
4	Conv2d-4	[1x64x27x27]	5x5	2	1	[1x192x27x27]
5	ReLU-5	[1x192x27x27]	-	-	-	[1x192x27x27]
6	MaxPool2d-6	[1x192x27x27]	-	0	2	[1x192x13x13]
7	Conv2d-7	[1x192x13x13]	3x3	1	1	[1x384x13x13]
8	ReLU-8	[1x384x13x13]	-	-	-	[1x384x13x13]
9	Conv2d-9	[1x384x13x13]	3x3	1	1	[1x256x13x13]
10	ReLU-10	[1x256x13x13]	-	-	-	[1x256x13x13]
11	Conv2d-11	[1x256x13x13]	3x3	1	1	[1x256x13x13]
12	ReLU-12	[1x256x13x13]	-	-	-	[1x256x13x13]
13	MaxPool2d-13	[1x256x13x13]	3x3	0	2	[1x256x6x6]
14	AvgPool2d-14	[1x256x6x6]	3x3	1	1	[1x256x6x6]
15	Dropout-15	[1x256x6x6]	-	-	-	[1x9216]
16	Linear-16	[1x9216]	-	-	-	[1x4096]
17	ReLU-17	[1x4096]	-	-	-	[1x4096]
18	Dropout-18	[1x4096]	-	-	-	[1x4096]
19	Linear-19	[1x4096]	-	-	-	[1x4096]
20	ReLU-20	[1x4096]	-	-	-	[1x4096]
21	Linear-21	[1x4096]	-	-	-	[1x1000]

TABLE I: Alexnet description

B. Getting the parameters of the analytical model

1) Collecting hardware related input

First we would gather input related to the hardware platform. Ideally, we aim to obtain all input data from readily available sources such as the hardware datasheets. In cases where the data is not specified (e.g., performance of specific CPU extensions), the data can be determined by running micro-benchmarks, such as MLC [24].

2) Collecting DNN model related input

Second, we collect input related to the DNN model. The Data Flow Graph (DFG) and the sizes of input and output tensors can be easily obtained from the model description. The algorithm used can be obtained from the description of the operation. For existing hardware like CPUs and GPUs, one can either explicitly select the algorithm (as in the case of MKLDNN) or use heuristics to decide the algorithm for implementation. For example, in NVIDIA GPUs, the Winograd approach is taken when the dimension of the weight tensor in convolution is 3×3 or 5×5. If tensor processing units are involved and the number of input channels is a multiple of 4, then the implicit GeMM-based convolution approach is taken, and so on. In the case of custom-designed futuristic hardware, developers will likely have some insight into the algorithm used. This information should be manually specified by the user to our analytical model.

3) Calculating the total number of fundamental operations

In order to calculate the fundamental operations of a given layer, the algorithm used to implement the layer is needed. Without the knowledge of the algorithm, the number of fundamental operations can vary drastically. To illustrate how fundamental operations are calculated, let us consider an

example. Consider the first layer of Alexnet in Table I which is a 2D Convolution (referred to as Conv-2D). For this particular layer, the dimension of the input tensor and the output tensor are $1 \times 3 \times 224 \times 224$ and $1 \times 64 \times 55 \times 55$ respectively, and a kernel of 11×11 with 64 filters is applied to perform the convolution. These input and output tensors are placed in $NCHW$ format where N, C, H, and W stand for the batch size, the channels, the height of the image and the width of the image, respectively. Similarly, the filter is placed in $KCRS$ format where K, C, R, and S stand for the number of filters, channels in the filter, filter height, and filter width, respectively. If we use the Direct Convolution algorithm, the computation involved is a seven layer nested loop (1 for batch size, 1 for input channels, 1 for output channels, 2 for kernels, 2 for images) with 1 MACC in the innermost nested loop.

This results in $1 \times 3 \times 64 \times (11 \times 11) \times (55 \times 55)$ MACC operations, where the last 4 values are drawn from the kernel size (11) and output tensor size (55).

All these parameters are mostly derived from the dimension of input, weight, and output tensors. However in some algorithms like Winograd based convolution, algorithm-specific parameters are involved which determine the number of MACC operations involved in the layer. In this algorithm, parameters called m (in case of GPUs for 3×3 it is 4) is used to determine the tile size on which transformations are applied.

Since the processing speed of the hardware is given in Floating Point Operations per second (FLOPS), we convert MACC operations into floating point operations. One MACC operation is equivalent to two floating point (addition and multiply) operations.

If information about the implementation is not known (in case of proprietary software), the number of floating point operations can be deduced by running hardware specific profilers. In this work, we use a priori computed analytical model for all the algorithms evaluated.

4) Accounting for the parallelism

The number of PEs used in a layer is typically bounded by the number of PEs in the hardware; DNNs are in general implemented by experienced programmers to maximize the parallelism that can be exploited from the hardware. However, there are some cases where the parallelism used by the algorithm is less than the parallelism available in the hardware (e.g., for better performance to avoid/reduce data replication). For example in case of batch normalization in GPUs, the parallelism depends on the number of input channels, and the number of channels can be less than number of PEs in some layers. Thus, our model determines the number of PEs used in a layer based on both the hardware and algorithm to appropriately account for the actual parallelism.

5) Calculating the data movement time

In order to calculate the number of bytes transferred, we take the following approach: Since the tensor sizes are large, the time spent in data movement can be reasonably estimated using number of bytes transferred through the interconnect (between memory and PE) and the interconnect bandwidth. Along with this, since the DNN layer algorithms are usually well optimized, the cache misses are mainly cold misses, with much lower capacity, conflict or coherence misses. **Hence the number of data elements transferred can be approximated by the size of the input and output tensors since first time accesses (cold misses) cannot be avoided.** This simplifies the data movement cost calculation to a great extent. Multiplying the number of data elements accessed with the datatype size (4 bytes for FP32) and then dividing it by the memory bandwidth will give the time spent in data movement. In section VI, one can see that this assumption does not sacrifice the accuracy.

It is important to consider that in some layers data gets transformed into another format before the actual computation is applied. For example in case of tensor core based convolution in GPU, data gets transformed from NCHW format to NHWC format before actual computation. After the computation, it gets transformed back to NCHW format. Since these transformations happen in distinct layers and no floating point operations are involved, the data movement cost has to be calculated explicitly for these layers.

6) Optimizations

Some frameworks optimize performance by employing techniques such as kernel fusion where two consecutive layers (or kernels) are merged so that data movement costs can be reduced. We have not explicitly modeled for all possible kernel fusions, but our approach to modeling the performance remains the same where the fused layer should be treated as a single layer in the DFG and steps 3-5 should be applied. If kernels A and B get fused to make a new kernel C, then the total data moved of kernel C will be the sum of data loaded in kernel A (input) and data stored in kernel B (output). The total number of fundamental operations of kernel C will be the sum of total number of fundamental operations in kernel A and kernel B.

V. Experimental Setup

In order to validate our analytical model, we evaluate several DNN models across a diverse set of hardware platforms.

Metric: We compare the execution time on the actual hardware to the execution time predicted by our analytical model and show the deviation of the predicted execution time to the actual execution time, expressed as a percentage, as a measure of accuracy. Positive values represent over estimation of execution time by our model. Similarly, negative values represent the under estimation.

Hardware: We run the DNN models across 4 platforms consisting of 2 CPUs and 2 GPUs (one with tensor cores and one without tensor cores). The specification of these platforms can be seen in Table II. For CPU platforms, we have used AVX ISA extension engines. These extensions are known for improving the Multiply and Accumulate operations, which are the fundamental operations in DNN models. However, when AVX engines in Intel CPUs are continuously used, the operating frequency is dynamically varied to prevent overheating, which prevents us from getting consistent evaluation results. Hence for consistency in our evaluation, frequencies are set to the lower range of operating frequency (800-1000 MHz and

Hardware Name	Intel(R) Xeon(R) Gold 6230 CPU	Intel(R) Xeon(R) CPU E5-2680 v3	Quadro RTX 8000 (TU102)	Ampere GA100 (40GB)	BLIMP system
Extension/ PE type	AVX	AVX	SMs	SMs and Tensor Cores	RISC-V cores
Number of PEs (P)	16	6	1 (72 SMs)	1 (108 SMs)	128
Processing Speed (S) of a PE	15.45 GFLOPS	21.64 GFLOPS	16.31 TFLOPS	19.49 TFLOPS (SMs) 155.92 TFLOPS (TCs)	500 KFLOPS (matrix-vector) 350 KFLOPS (matrix-matrix)
Interconnect Bandwidth	77.8 GBPS	15.334 GBPS	672 GBPS	1555 GBPS	39.936 GBPS

TABLE II: Hardware specifications.

1200-1400 MHz for Intel Xeon Gold based machines and Intel Xeon E5-2680 based machines, respectively). For GPUs, we have used an A100 (40 GB) GPU to validate tensor core based DNN model execution and Quadro RTX 8000 GPU to validate non-tensor based DNN model execution.

Software: From the software side, for CPUs, we have used benchDNN [23] to run specific kernels of the DNN model. BenchDNN provides the average execution time after running the kernel multiple times, thus reducing deviations that can happen from one run to the next. For GPUs, we have used pytorch [19] to run the DNN model. Nsight compute [25] is used to obtain the execution time of the DNN model.

DNN Models: We have taken six well known models Alexnet [26], Caffenet [27], Resnet18 [28], Resnet50 [28], Vgg16 [29], and Yolonet [30]. Based on the number of layers and the floating point operations, these DNN models represent small, medium and large DNN models. This provides diversity from the DNN model side to check the robustness of our model.

Input Parameters to analytical model: Our analytical model requires the FLOPS rating, memory bandwidth, and the number of processing engines to compute the execution time. In CPUs, since CPU FLOPS ratings for a particular frequency under a particular ISA extension (AVX) is not readily published in datasheets, we have obtained it by running sample microbenchmarks for given configurations. We have obtained the bandwidth numbers using the Memory Latency Checker (MLC) [24] for given configurations. On CPUs, we have used a batch size of 1, since it saturates the CPU pipeline. In case of GPUs, we have used the specification sheets [21], [22] to get the FLOPS rating and the memory bandwidth numbers. Nsight compute [25] is used to obtain the the operating frequency of the GPU. On GPUs, we have used a batch size of 128, which provides good GPU utilization across all DNN models and GPUs.

VI. Evaluation

A. Accuracy of the model

In Figure 3a, we can see that the deviation of the predicted execution time to the actual execution time. All deviations are within 7.5% to -15.89% deviation from the actual performance on the hardware, suggesting that the model can capture performance fairly well to be useful for different purposes as investigated below. In this result, we have considered the parallelism modifications and the inefficient kernel modifications mentioned in the previous section. In case of CPUs, the deviation is with in -1.65% to -15.89% for Xeon(R) Gold 6230 CPU and is with in 7.5% to 1.45% for Xeon(R) CPU E5-2680 CPU. The deviation is with in 1.52% to -7.52% in case of Quadro RTX 8000 GPU. In case of Ampere GA100 GPU, the deviations are with in -9.82% to -14.21%. *The results show that our simple analytical modeling approach is reasonably accurate and robust across various diverse hardware platforms for small, medium and large DNN models.*

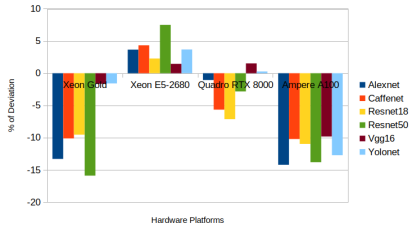
B. Accuracy of modeling convolution algorithms

One of the main features of our analytical model is modeling algorithms based on fundamental operations. In case of convolution, which are compute intensive, we have variety of algorithms to choose from to implement a given layer on GPUs. Convolution layers take a significant portion of DNN execution time. In case of Quadro RTX GPU, convolutions take 61% to 83% and in case of A100 GPU, they take 55% to 80% of the total execution time for previously mentioned DNN models. Hence modeling them accurately improves the accuracy of our model. The accuracy of our analytical model for various convolution algorithms across GPUs can be seen in Figure 3c. Some algorithms are not available in all hardware platforms. Since this variety of algorithms are not available in CPUs (oneDNN library) we have done comparison only for GPUs. In case of our Quadro RTX GPU, for widely used algorithms like Winograd and Direct convolution algorithms, the deviation is within 5%. For the A100 GPU, the deviation is within -10% for Direct and tensor based convolution algorithms. FFT based approach is has a deviation of -17.4% in case of Quadro RTX GPU. But this approach is taken only for few layers in Resnet18 and Resnet50. Hence doesn't affect the performance much. But it has a deviation of -2.3% in case of A100 GPU. *Thus we can conclude that our analytical model accurately models various convolution algorithms.*

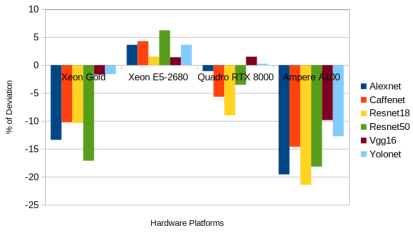
C. Handling inaccuracies : Parallelism and Inefficient kernels

There are 2 main reasons for inaccuracies in our analytical model's estimation: (i) kernels fail to completely exploit the parallelism offered by the hardware; (ii) even if the kernels exploit all the hardware resources, if an inefficient kernel implementation gets chosen, then it can cause the deviation. Without accounting for these factors, the results can be more inaccurate, as shown in Figure 3b. Although there is no way to account for inefficient kernels in our analytical model, parallelism factor can be accounted.

Batch Normalization in GPUs: In case of the Batch Normalization layer (which is used in Caffenet, Resnet18, and Resnet50) in GPUs, the parallelism (number of CUDA blocks launched) is parameterized by the number of input channels. In the initial layers of Resnet18 and Resnet50, there are around 64 channels, which is less than the number of PEs present in the GPUs. Hence the layer fails to completely exploit the parallelism offered by the hardware. *In this case, we reduce the*



(a) For DNN models across hardware platforms



(b) For DNN models across hardware platforms without considering parallelism and inefficient kernel effects

(c) Algorithm-wise deviation in GPUs

Fig. 3: Deviation in predicting the execution time.

number of processing engines to the number of input channels in the input tensor. When accounting for this, in the first batch normalization layer of both Resnet18 and Resnet50 (they are of same dimension), in case of A100 GPU, the estimated time changed from 0.93 ms to 1.55 ms and in case of Quadro RTX 800 it changed from 1.949 ms to 2.14 ms. Both of these changes improve the accuracy of our model. *In summary, while building an analytical model, accounting for parallelism at the kernel level will help to improve the accuracy of the model.*

D. Use Case 1: Performance Debugging (Pinpointing inefficient kernels)

To some extent, our model captures the expected performance of these models. When its deviation is high, developers can use this information to find inefficient kernels. This can be illustrated with the following example: In case of Fully connected layers in Alexnet and Caffenet on A100 GPUs, pytorch picks the $ampere\_sgemm\_32x32\_sliced1x4\_tn$ kernel. However, this implementation is less efficient for the given input dimension as it is unable to exploit the full parallelism due to *tail effects*, which are defined as the resource imbalance leading to low utilization during the last wave of computation. However, when we use cuBLAS based $ampere\_sgemm\_64x32\_sliced1x4\_nn$ and $splitKreduce\_kernel$ kernels for the same input, it achieves full parallelism better than the kernel chosen by pytorch. This may happen as the developers might not have tested the fully connected layer for this set of dimensions and chosen the kernel based on heuristics. Without the analytical model, it would be hard to find this deviation. By doing this switch to a more efficient algorithm for this particular layer and combination of inputs, the execution time reduces from 1.42 ms to 1.08 ms, which is close to our estimation of 1.02 ms. This illustrates one important (performance debugging) use case for our analytical model.

E. Use Case 2: Predicting for Hypothetical/Future Hardware

	Input size	Output size	Kernel size
Kernel 1 (K1)	[1x256x28x28]	[1x512x28x28]	3x3
Kernel 2 (K2)	[1x512x7x7]	[1x1024x7x7]	3x3

TABLE III: BLIMP convolution kernels.

In order to show the validity of our model on futuristic hardware, we have considered a Processing In Memory (PIM) accelerator called BLIMP [31], [32]. In this accelerator, a RISC-V core is incorporated at the bank level in a DDR-DRAM chip. The hardware provides high bandwidth along with higher degree of parallelism solving the memory bottleneck problem.

Kernels and algorithms: We have considered 2 convolution kernels, which are mentioned in Table III (with appropriate padding) for evaluation. Both kernels are run using the Im2Col and Winograd algorithms. Our model enables us to determine the appropriate algorithm to be deployed on this accelerator.

Input parameters: The hardware parameters are specified in Table II. Since these algorithms are written by ourselves not by expert library programmers, we have chosen 2 processing speeds for matrix-matrix multiplication (for dot product computation) and matrix-vector multiplication (for GeMM kernels) to account for differences in speed. In case of CPUs and GPUs, the libraries are written in a way to achieve the peak performance offered. Hence this change is unnecessary for CPU and GPU implementations.

	Im2Col Approach			Winograd Approach		
	BLIMP time(s)	Estimated Time(s)	Dev%	BLIMP time(s)	Estimated Time(s)	Dev%
K1	33.057	28.994	-12.29	12.223	13.389	9.54
K2	10.085	8.738	-13.36	3.734	4.257	14.01

TABLE IV: BLIMP convolution results.

Results: We employ the simulation framework mentioned in [31]. From Table IV it can be seen that for the Im2Col approach, the deviations are -12.29% and -13.36%, and for the Winograd approach, the deviations are 9.54% and 14.01%. Based on the execution times in Table IV, it can be seen that in both cases, the Winograd approach is better than the Im2Col approach. In addition this, the speed up deviation in comparison to Intel Xeon Gold CPU are 14.01% and 15.42% for Im2Col approach and -8.71% and -12.29% for Winograd approach. *Thus our analytical model in addition to execution time, also accurately predicts the speed up offered by the BLIMP accelerator.* In this way, our analytical model can be used to predict which algorithm should be used on a given hardware.

VII. Conclusion

In this paper, we identify three primary factors impacting DNN performance and show how using these factors can create reasonably accurate analytical performance models that are fast and applicable to both current hardware and hypothetical hardware that may not exist. The first factor (compute) is based

on the number of floating point operations, which requires analyzing DNN algorithms to see the impact of the input tensor sizes and DNN parameters. The second factor (data movement) is based on the input/output tensor sizes, where we find that caching is good and most of the data movement is from cold misses when accessing the input/output for first time. The third factor (parallelism) is based on the number of processing elements within the hardware as well as the degree of parallelism used in DNN algorithms. Our evaluation shows how performance models built on these three factors are fast and reasonably accurate ($\pm 10-15\%$) across many DNNs and hardware types including GPUs, CPUs, and a futuristic Processing In Memory (PIM) accelerator called BLIMP.

ACKNOWLEDGMENT

We thank Adithya Kumar for his help in providing valuable suggestions. This research was supported in part by National Science Foundation grants 1714389, 1912495, 1909004, and 2211018.

REFERENCES

[1] Y. Ren, S. Yoo, and A. Hoisie, "Performance analysis of deep learning workloads on leading-edge systems," in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2019, pp. 103–113.

[2] U. U. Hafeez and A. Gandhi, "Empirical analysis and modeling of compute times of cnn operations on aws cloud," in *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 181–192.

[3] S. A. Mojumder, M. S. Louis, Y. Sun, A. K. Ziabari, J. L. Abellán, J. Kim, D. Kaeli, and A. Joshi, "Profiling dnn workloads on a volta-based dgx-1 system," in *International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 122–133.

[4] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko, "Benchmarking and analyzing deep neural network training," in *International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 88–100.

[5] S. Xi, Y. Yao, K. Bhardwaj, P. Whatmough, G.-Y. Wei, and D. Brooks, "Smaug: End-to-end full-stack simulation infrastructure for deep learning workloads," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–26, 2020.

[6] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv:1811.02883*, 2018.

[7] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 553–564.

[8] S. J. Plimpton, S. Agarwal, R. Schiek, and I. Richter, "Crosssim," Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2016.

[9] L. Bai, W. Ji, Q. Li, X. Yao, W. Xin, and W. Zhu, "Dnnabacus: Toward accurate computational cost prediction for deep neural networks," *arXiv preprint arXiv:2205.12095*, 2022.

[10] D. Justus, J. Brennan, S. Bonner, and A. S. McGough, "Predicting the computational cost of deep learning models," in *IEEE international conference on big data (Big Data)*. IEEE, 2018, pp. 3873–3882.

[11] E. Gianniti, L. Zhang, and D. Ardagna, "Performance prediction of gpu-based deep learning applications," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2018, pp. 167–170.

[12] Y. Gao, X. Gu, H. Zhang, H. Lin, and M. Yang, "Runtime performance prediction for deep learning models with graph neural network," in *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2023, pp. 368–380.

[13] F. Jia, D. Zhang, T. Cao, S. Jiang, Y. Liu, J. Ren, and Y. Zhang, "Codl: efficient cpu-gpu co-execution for deep learning inference on mobile devices," in *International Conference on Mobile Systems, Applications and Services*. ACM, 2022, pp. 209–221.

[14] Z. Wang, P. Yang, B. Zhang, L. Hu, W. Lv, C. Lin, C. Zhang, and Q. Wang, "Performance prediction for deep learning models with pipeline inference strategy," *IEEE Internet of Things Journal*, 2023.

[15] X. Y. Geoffrey, Y. Gao, P. Golikov, and G. Pekhimenko, "Habitat: A runtime-based computational performance predictor for deep neural network training," in *Annual Technical Conference*. USENIX, 2021, pp. 503–521.

[16] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *International Conference on Learning Representations*, 2016.

[17] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 4013–4021.

[18] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A gpu performance evaluation," *arXiv*, 2014.

[19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[20] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[21] "NVIDIA A100 GPU Specification sheet," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf, 2020, [Online; accessed 31-July-2023].

[22] "NVIDIA Quadro RTX 8000 GPU Specification sheet," https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/quadro-rtx-8000-us-nvidia-946977-r1-web.pdf, 2019, [Online; accessed 31-July-2023].

[23] "BenchDNN benchmark in OneDNN," https://github.com/oneapi-src/oneDNN.

[24] "Memory Latency Checker Tool," https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html, 2021, [Online; accessed 31-July-2023].

[25] "NVIDIA Nsight Compute," https://developer.nvidia.com/nsight-compute.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[27] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the ACM International Conference on Multimedia*, 2014, pp. 675–678.

[28] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[29] K. Simonyan and A. Zisserman, "Very deep convnets for large-scale image recognition," *Computing Research Repository*, 2014.

[30] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[31] S. B. Rai, A. Sivasubramaniam, A. Kumar, P. V. Rengasamy, V. Narayanan, A. Akel, and S. Eilert, "Design space for scaling-in general purpose computing within the ddr dram hierarchy for map-reduce workloads," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2021, pp. 113–123.

[32] A. Devic, S. B. Rai, A. Sivasubramaniam, A. Akel, S. Eilert, and J. Eno, "To pim or not for emerging general purpose processing in ddr memory systems," in *Proceedings of the International Symposium on Computer Architecture*, 2022, pp. 231–244.