Hybrid PTX Analysis for GPU accelerated CNN inferencing aiding Computer Architecture Design

Christopher A. Metz[𝕌] Christina Plump[‡] Bernhard J. Berger[†] Rolf Drechsler[𝕌,‡]
[‡]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
[𝕌]Institute of Computer Science, University of Bremen, Bremen, Germany
[†]Institute of Embedded Systems Hamburg University of Technology Hamburg, Germany
{cmetz, cplump, drechsler}@uni-bremen.de; bernhard.berger@tuhh.de

***Abstract*—*General-Purpose Computation on Graphics Processing Units* (GPGPUs) are becoming crucial in accelerating computing capacity. Due to the massive parallelism capabilities of GPUs, they can achieve impressive speedups of up to 32 times compared to common CPUs. However, writing highly parallel code and utilizing a GPU is challenging for programmers. Developers are facing new challenges since GPUs handle threads and parallelism differently from CPUs. Academia and industry proposed several profilers to support developers in terms of code optimization. These profilers often require an actual device (e.g., GPU) and take a long time for the profiling process.**

We propose HyPA, a hybrid *Parallel Thread Execution* (PTX) Analyzer that inspects PTX code statically and dynamically. HyPA implements a partly functional emulator that executes instructions that rely on runtime dependencies to count the number of executed PTX instructions and divergent branches. HyPa executes compiled kernels—the programs that run on GPUs—generated by the CUDA compiler and supports the full PTX 7.7 specification. Our functional emulator allows significantly faster analysis of PTX code compared to standard profilers. In our evaluation, we quantify this increase in performance through benchmark runs. HyPA achieved speedups of up to 536% compared to the nvprof profiler. Moreover, our approach can gather performance metrics beyond static analysis (e.g., branch efficiency) by a faster execution time than by profiling the application on an actual device. Finally, we provide an open-source implementation of HyPA to help developers and system designers in further research and development.

***Index Terms*—GPU, CUDA, PTX, Power and Performance Optimization**

I. Introduction

The usage of general-purpose applications on *Graphics Processing Units* (GPUs) increased over the last decades. Nowadays, their applications range from embedded devices [1] to supercomputers like the Summit [2], covering almost all kinds of electronic devices.

While in June 2015, nearly 19% of the systems on the TOP500 list used GPU accelerators, the proportion increased to nearly 30% in June 2022 [3]. The use of GPUs for training and inferencing of *Deep Neural Networks* (DNN) benefits this increase [2], [4].

GPUs (e.g., NVIDIA V100) offer 32 times better performance on DNNs compared to classical CPUs [5]. For example, by utilizing the potential of 256 GPUs' parallelism, resnet50 [6]—a *Convolutional Neural Network* (CNN) with 50 hidden layers—can be trained on the full ImageNet [7] data set in one hour [8]. In contrast, when resnet50 was introduced by [6] the training took 29 hours [9].

Besides all their apparent merits, GPUs' drawback is their high power consumption to achieve their high levels of performance. For example, the Summit supercomputer uses 27,648 NVIDIA Volta GPUs. Its energy consumption is 13 million watts [2]. Literature estimates that a 5% decrease in power consumption of the Summit will lead to a saving of 1 million dollars [10]. In *Internet of Things* (IoT) devices, a higher power consumption—due to the usage of *Machine Learning* (ML) algorithms—has been reported [1] as well. For instance, executing object recognition locally on an Nvidia Jetson TX1 consumes 7 Watts, compared to 2 Watts, when the object recognition is offloaded to the cloud [1]. Therefore, decreasing power consumption is a relevant factor for IoT devices as well. With the application of ML in many areas of electronic devices, the energy efficiency of ML algorithms becomes crucial.

Three strategies are possible in order to achieve energy-efficient applications 1) designing more energy-efficient devices, 2) optimizing the applications' implementation or 3) optimizing the chosen device for the chosen application. Options 1 and 2 are usually long-term approaches, as pursuing the second strategy requires deep inside knowledge of the code and behavior of an actual device to maximize its utilization, and regarding option 1, utilizing a GPU energy-efficiently is a complex and challenging task [11], [12]. However, the third option may allow users to supply only the performance necessary for their task and thus, optimize energy efficiency. To do so, one needs information about the potentially employed devices, e.g. the different GPUs, for his specific task, e.g. a CNN-inferencing task. Standard metrics that are used for the prediction of power consumption and performance are the *branch-efficiency*, the *#instructions*, and the *#floating-point-operations* [13], [14]. Code profiling and execution analysis need to be performed for all possible setups to gather these metrics.

Academia and industry proposed different approaches for code profiling and analysis. The three main approaches—which we will describe in the following—are 1) classical profiler tools that measure performance counters during the execution of the application on an actual device, 2) simulators that simulate or emulate the actual device and 3) static code analysis where performance and power prediction is calculated based on the application's source code. However, all approaches have their limitations, discussed in the following:
(1) Profiling tools like nvprof [15], CUPTI [16], or nsight[1] log different performance counters (e.g., the total number of executed instructions) during execution on an actual GPU [15], [17]–[19]. To find the optimal GPU, it's necessary to run the profiling process on multiple GPUs since the profiling results are restricted to the GPU in use. This leads to three main disadvantages of profilers.

- Performance counters are not consistent across different GPUs. This means, that the same performance counter can be calculated differently on different GPUs or the corresponding counter may not exist. Hence, comparing these metrics can be challenging or even impossible.
- The profiling step is very time-consuming as its duration is significantly longer than the application run-time itself due to its dependence on the GPU instead of the application [17],

Special thanks go to Torben Eberman, who supported the software development.

[1]https://developer.nvidia.com/nsight-systems

[18], [20]. Hence, profiling GPUs is not a time-efficient way of determining the most appropriate GPU.

- For profiling, access to the actual GPU is necessary. As results are not transferrable between different GPUs due to the dependence on their machine language [18], the user would need access to every GPU that is evaluated, which is very costly.

(2) Simulation techniques solve the abovementioned availability problem of GPUs. They allow the computation of metrics without actual access to the GPU. In the past, tools like GPGPUSim [21], and Ocelot [22] were proposed. However, since they emulate GPU behavior and execute the application on CPUs, which do not offer the same high parallelism capabilities, the execution time takes much longer than the profiling on actual GPUs. Additionally, they do not achieve identical results compared to profiling. Hence, while helping to reduce costs, simulation techniques are less favourable in terms of time than profiling techniques.

(3) Another possibility is static code analysis. Its main drawback is the fact that it can not analyze dynamic behavior. Conditional jumps may have different control-flow paths in different threads which can only be determined with a dynamic analysis. Therefore, the result of static analysis either underestimates (in the case of loops) or overestimates (in the case of non-entered ELSE branches) several metrics (e.g. numbers of instructions). Other metrics, like branch efficiency, are inherently based on dynamic analysis, and can therefore not be computed. Its advantage, however, is the small time consumption [20], [23]–[25].

The three existing methods have crucial drawbacks for gathering standard metrics to predict power and performance based on ML. Therefore, we propose an approach for computing the respective metrics, which combines the time-related advantage of static analysis with the advantage of simulations without having its time-related drawback. Our approach is based on the idea that it is not necessary to simulate the entire application on the GPU, but only those parts that influence conditional jumps and may therefore obfuscate the results of static analysis. The resulting hypothesis is twofold: First, a hybrid analysis approach is more time-efficient than the standard simulation approach and second, not only more precise than a static analysis but capable of determining metrics that are inherently based on dynamic analysis.

Thus, we created an *Hybrid PTX Analyzer* (HyPA) that considers dynamic dependencies within the code and can be used without executing the entire application (e.g., CNNs). The basic idea of our tool is to read all instructions, consider the number of threads, and search for dynamic dependencies. Afterwards, a functional simulator executes only those PTX instructions on a CPU that rely on dynamic run-time dependencies for conditional jumps. By this, HyPA is overcoming the lack of speed of existing GPU simulators. The generated profiles can be used for power and performance prediction of GPU applications (e.g., [26]) or for better code understanding (e.g., [18], [20], [21]). HyPA gives detailed information on the number and type of instructions, floating point operations, and divergent branches.

Our main contributions can be summarized as follows:

1) A hybrid approach of PTX emulation to profile CUDA applications on a low-level code basis
2) An automatic extraction of the following PTX code metrics: number of instructions, floating point operations, number of divergent branches, and branch efficiency
3) An implementation of HyPA as an opensource project to help developers, computer architects, and researchers in their work

This paper is structured as follows: Section II provides information regarding GPUs and PTX. We describe our approach in Section III. The experimental results are shown in Section IV and discussed in Section V. Section VI covers related work. Lastly, Section VII concludes the paper.

II. Background

In this section, we explain the necessary background and introductory concepts of General Purpose GPUs and PTX ISA which are necessary to understand the proposed approach and to make the paper self-contained.

A. General Purpose GPU

GPUs have a different and more complex architecture compared to traditional CPUs. The GPU architecture consists of a scalable number of *Streaming Multiprocessors* (SMs). An SM is partitioned into multiple *Processing Units* (PUs) to improve the utilization of GPUs. In the case of the NVIDIA V100, an SM is divided into four PUs. Each PU contains 16 *Floating Point* (FP) 32-Bit Cores, 8 FP64 Cores, 16 INT32 Cores, one Tensor Core (with mixed-precision Tensor Cores for Deep Learning), an L0 instruction cache, one warp scheduler, one dispatch unit, and a 64KB Register File [25], [27].

Kernels compiled for a GPU are subdivided into *Cooperative Thread Arrays* (CTAs), also called thread blocks [19]. A CTA is further divided into groups of 32 threads called a warp [25], [28]. All threads inside a warp execute the same instruction [20]. This principle is similar to *Single Instruction Multiple Data* (SIMD). However, NVIDIA calls it *Single Instruction Multiple Threads* (SIMT). Unlike SIMD instructions, the concept of warps is new to many programmers. Additionally, they only have control over the total number of threads but not over the warp handling. Thus, programmers write a program for one thread and then specify the number of parallel executions of this thread [25].

All threads in one warp are executed on one SM. An SM can also handle multiple warps [19], [28]. The number of concurrently running warps is determined by the resource requirements of each warp, such as the number of registers or shared memory usage. Beginning with the Volta architecture, the warp-synchronous programming, that all threads executing the same instruction within a warp, has been made obsolete [19]. As a result, the generated threads are less ideal due to divergent branches.

The analysis in this paper considers primarily Nvidia GPU architectures starting from Volta and their successors because they are provided with a solid and more reliable theoretical foundation since there are massive changes between the Volta architecture and their predecessor [19].

B. Parallel Thread Execution

Nvidia offers the *Compute Unified Device Architecture* (CUDA) library to run applications on their GPUs. Frameworks like Tensorflow include CUDA so users can easily run their applications on GPUs [29]. Therefore, the code is compiled to *Parallel Thread Execution* (PTX) to execute the CUDA applications on a GPU. PTX is an *Instruction Set Architecture* (ISA) including memory access and computational instructions (e.g., ADD, MUL, FMA) and is translated to native binary micro-instructions [30]. Some PTX instructions cannot be translated to a single binary micro-instruction and are therefore represented by multiple ones [25]. Nvidia provides a detailed natural language description of the PTX model which gives developers a detailed understanding of PTX's syntax and behavior [2].

[2]https://docs.nvidia.com/cuda/parallel-thread-execution/

The binary micro-instruction language (SASS) is the target machine language of a specific GPU [31] PTX is designed as a virtual ISA to be portable between different GPU generations with different instruction sets (i.e., SASS implementations). The lack of portability makes SASS unattractive for analysis [18] as it limits the analyses to a single GPU.

The number of executed instructions of a PTX code can be calculated statically by counting the total instructions per thread block and multiplying it by the total number of threads. However, this only includes the static number of instructions. Since PTX supports conditional jumps, the actual number of executed instructions might differ from the results of this static analysis [25]. The code needs to be executed to determine data dependencies for conditional jumps to get more accurate results. Static analyses, e.g. [25], [32], ignore underlying data dependencies. Consequently, the counted number of instructions is only an approximation of the actual number. On the one side, the counted number of instructions can be an overestimation if instructions are not executed due to conditional jumps. On the other side, the counted number can also be an underestimation, if the PTX code includes loops and parts are executed multiple times due to conditions.

***Divergent Branches**:* The PTX ISA and NVIDIA's GPU architectures allow so-called divergent branches. A divergent branch is a branch where some threads are within the same warp branch while others are not (cf. Figure 1). Consequently, the number of divergent branches is always smaller or equal to the number of branches. If all threads in a warp take the same path in the control flow no divergent branches occur. Deferring the concept of SIMT, not all threads execute the same instruction for the occurrence of divergent branches [20], [23], [24]. This behavior leads to an unknown amount of instructions that cannot be detected by static code analysis. Since these instructions rely on conditional jumps, the register values must be calculated. As a consequence, a simulation of the program is necessary to determine the value of the registers that are checked at the branch conditions.

Two cases of divergent branches can occur 1) IF without ELSE; during this case, some threads enter the IF and execute the additional instructions while others are idle due to the SIMT concept where all threads have to execute the same instructions. 2) IF, with ELSE, this case is more complex; while some threads enter the IF and execute the instructions, other threads are idle. Afterwards, when the previously idle threads enter the ELSE statement, the IF-threads become idle [20]. Both cases may lead to a different number of executed instructions; hence, counting the instructions of a PTX code and multiplying them by the number of threads will not lead to the correct number of executed instructions.

III. Hybrid PTX Analysis

The PTX Code analysis is split into three parts 1) static code analysis and dependency detection, 2) dynamic code analysis and emulation, and 3) profile generation. The general workflow of HyPA is illustrated in Figure 2.

A. Static PTX Analysis

HyPA starts with a static code analysis to detect all instructions that need to be executed, the number of threads that will be raised, and the run-time dependencies that need simulating to resolve conditional jumps based on dynamic dependencies. Therefore, HyPA is parsing the PTX Assembler and stores the information in an intermediate

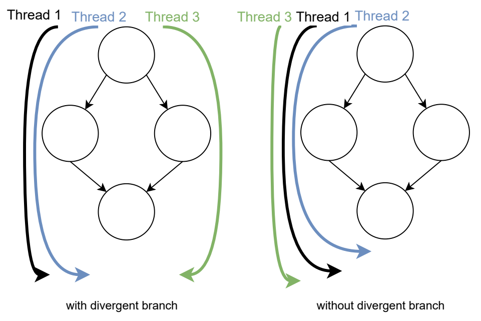

Fig. 1: Example of three different threads taking their path in a control flow graph. In the left graph thread 3 creates a divergent branch as in the right graph no divergent branches are created.

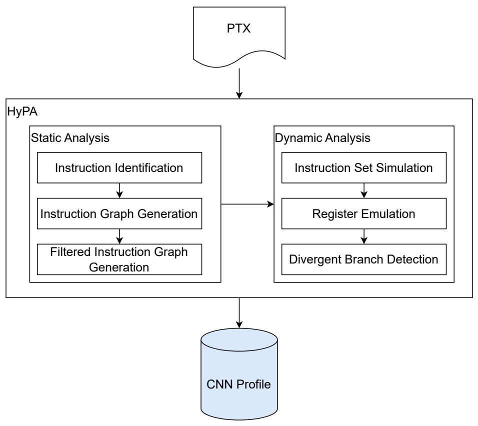
PTX
HyPA
Static Analysis
Instruction Identification
Instruction Graph Generation
Filtered Instruction Graph Generation
Dynamic Analysis
Instruction Set Simulation
Register Emulation
Divergent Branch Detection
CNN Profile

Fig. 2: General Workflow of the PTX Analyzer *HyPA*

representation. In this intermediate representation, an instruction is formally denoted by a vector

$$ins = (id, dr, sr_1, ..., sr_m, CTAid, CBid) \tag{1}$$

where id stands for the unique instruction identifier, dr and $sr_j, 1 \leq j \leq m$ denote the destination and source registers, resp., and $CTAid, CBid$ are the CTA-Id and the Code-Block-Id. INS is the set of all possible instructions which enables the denotation of $\mathcal{I}(P) \in INS^n$ for the list of all instruction lines of a given program p. $\mathcal{I}_i, 1 \leq i \leq n$ denotes the i-th instruction line in this denotation scheme. Thus, the formal notation of the exemplary PTX code line `add.s64 %rd48 %rd7, %rd47;` is $ins = (0, rd48, rd7, rd47, 0, 0)$. Since it is the first identified PTX line, the ID id is set to zero (0). Moreover, since no CTA and labels are specified in the example, both CTA-Id and Code-Block-Id also are set to zero (0) in this case. Please note, that the maximum number of specified source registers is limited to 4 based on the PTX documentation [30].

In the following, we will shortly define dependency graphs [33] and their restriction to conditional instructions.

Definition 1 (Dependency Graph)**.** *A Dependency Graph G is defined as a tuple $G = (V, E)$, where V is a finite set of nodes, denoting instructions, and $E = \{(v_1, v_2)|v_1, v_2 \in V\}$ is a set of directed edges. An edge from node v_1 to node v_2 indicates that the latter instruction is dependent on the former.*

Each PTX code line containing an instruction ($\mathcal{I}_i$) is represented by a node ($v_i \in V$) where all instructions included in a CTA lead to a

Dependency Graph G_{CTAid} containing these instructions as well as their data dependencies. A dependency of two or more instructions exists if a source register (sr_j) of instruction $\mathcal{I}_i$ is the destination register (dr) of earlier occurring instructions. Consequently, the node v_i for instruction $\mathcal{I}_i$ depends on the nodes v_l whose instructions $\mathcal{I}_l$ have previously been written to the source registers specified for $\mathcal{I}_i$ and thus an edge $e = (v_l, v_i)$ is included in the graph for each instruction that does so [34].
During Dependency Graph (G) construction all jump instructions are stored in a *Control Flow Instruction List* (CFIL). At the same time, these nodes constitute the set $V_{branch} \subseteq V$, namely all nodes in the instruction graph that may affect a specific jumping operation. The jump instructions can be identified unambiguously based on the identifiers (id) assigned to the lines during initial parsing,

Filtered Dependency Graph: Based on our initial hypothesis, not all instructions must be executed to decide whether a thread will branch. Therefore, a dynamic slicing [34] is performed. Starting from the jump condition of a specific code block, all influencing nodes are traced back through the instruction graph. The resulting subgraph is called *Filtered Dependency Graph* (FDG) in the following. We formally define this in the following

Definition 2 (Filtered Dependency Graph (FDG))**.** *The Filtered Dependency Graph is defined as subgraph $G_{v^*} = (V', E')$ of $G = (V, E)$, where $v^* \in V_{branch} \subseteq V$, with*

$$\begin{aligned} V' = &\{v_0 \in V | \exists \pi = v_0 v_1 ... v_n \ with \ v_n = v^*, \\ &(v_{i-1}, v_i) \in E, \ \forall i \in \{1, ..n\}\} \subseteq V \\ E' = &\{e = (v_1, v_2) \in E | v_1, v_2 \in V'\} \end{aligned}$$

The FDG only consists of the instructions that need to be executed to identify if a conditional jump operation will be triggered or not. Hence, all jump operations have to be located to generate the FDG. Based on the CFIL from the dependency graph generation, the paths to the root nodes are identified. The working principle is as follows: A jump instruction from the CFIL marks the starting point at node (A) in the dependency graph (G). This starting node (A) is added to the FDG. For every added node, its (transitive) parents are added as well. The resulting graph trivially is a subset of G. The identified FDG will then be simulated during the Dynamic PTX Analyzation.

B. Dynamic PTX Analyzation

Based on the FDGs which are generated for all CTAs, the dynamic PTX analysis proceeds by emulating the GPU. The emulation does not perform the full application (i.e., PTX code) since the FDG only consists of a subset of instructions. Every generated FDG is given to the *Instruction Set Simulator* (ISS) which builds the core of the dynamic PTX analysis.

Instruction Set Simulator: Based on a C++ CPU implementation, the ISS executes PTX instructions included in the FDG. The necessary registers are emulated based on a symbol table, where each register is identified by its name and receives the belonging value after the instruction emulation. Moreover, the ISS receives the instructions, and based on a fixed assignment, the equivalent C++ implementation is performed. We reimplemented nearly all existing PTX instructions in C++, allowing the analyzer to apply to all CUDA applications.

Register Emulator: To correctly determine the jump conditions, it is necessary to save the values of the registers. Hence, we designed a data structure consisting of a key-value pair that can be defined as a function based on the definition in Eq. 2.

$$f : K \to V \tag{2}$$

TABLE I: An overview of CNN models used in the experiments

Model name	Input Size	Layers	Neurons
m-r50x1	224×224	50	15,903,016
m-r50x3	224×224	50	143,111,080
m-r101x3	224×224	101	25,3408,168
m-r101x1	224×224	101	28,158,248
m-r154x4	224×224	154	611,981,544
resnet101	224×224	101	55,886,036
resnet152	224×224	152	79,067,348
resnet50v2	224×224	50	31,381,204
resnet101v2	224×224	101	51,261,140
resnet152v2	224×224	152	75,755,220
nasnetmobile	224×224	771	27,690,705
nasnetlarge	331×331	1041	290,560,171
densenet121	224×224	121	49,926,612
densenet169	224×224	169	60,094,164
densenet201	224×224	201	77,292,244
mobilenet	224×224	28	16,848,248
inceptionv3	299×299	48	32,554,387
vgg16	224×224	16	15,262,696
vgg19	224×224	19	16,567,272
efficientnetb0	224×224	240	25,117,095
efficientnetb1	240×240	342	40,150,331
efficientnetb2	260×260	342	50,908,981
efficientnetb3	300×300	387	87,507,971
efficientnetb4	380×380	477	180,088,531
efficientnetb5	456×456	579	358,290,427
efficientnetb6	528×528	669	605,671,091
efficientnetb7	600×600	816	1,046,113,195
Xception	299×299	71	62,981,867
MobileNetV2	224×224	53	21,815,960
InceptionResNetV2	299×299	164	81,201,907
alexnet	227×227	8	650,000

The required keys are determined based on the filtered instruction graph, and each key occurs precisely once. If the ISS now calculates a value, it is assigned as a value to the corresponding register key. This process is defined as follows:

$$f(k_i) \leftarrow v_{new} \tag{3}$$

The currently assigned value v_i is overwritten with the new value v_{new} if a register k_i is written to several times during the emulation.

In addition, the ISS reads the values of a register specified as source register for instructions from this data structure $f(k_i) = v_i$. This guarantees that during the emulation of the PTX code, the correct values are always present in the register emulator.

Divergent Branch detection: Each time a jump is performed, the current code-block ID is stored in a list. Each thread has its list of code-block IDs. The order of the IDs in the code-block ID list indicates the exact program path of the respective thread.

If the lists of all threads of a CTA are compared to each other, divergent program paths (e.g., divergent branches) can be recognized, and threads, which deviate, can be identified. When the code-block ID list is created, the first code block is always given the ID zero (0). Consequently, all code block ID lists start with ID zero (0), followed by the respective thread's code-block- ID sequence. Different orders of code-block IDs identify different code paths and thus divergent branches. If no divergent branch occurs, all code path lists have the same order of code-block IDs.

C. Profile Generation

After the analysis, a PTX profile in the form of a *Comma-Separated Values* (CSV) file is generated. This file can be used for power and performance prediction and other optimization tools. All analyzed PTX files are combined into a single output file, which includes details such as filename, PTX version, PTX target, PTX address size, file instructions, CTAs, thread count, static instructions count, dynamic instruction count, FP instructions count, executed instructions count, divergent branches, divergent branches for each CTA, branch efficiency, and duration (in milliseconds).

IV. Experimental Setup and Results

In order to evaluate our hypothesis that the hybrid approach combines the advantages of static analysis and simulation approaches, while avoiding their disadvantages, we aim to answer three research questions with our evaluation.

Research Question 1. *How does the run-time of the profiling approach compare to the hybrid approach?*

Research Question 2. *How much do the acquired metrics of the hybrid approach differ from the profiling approach?*

Research Question 3. *Can the hybrid approach measure metrics that the static analysis can not and how do they compare to the profiling approach?*

We base our research questions on the profiling approach instead of the simulation approach due to three reasons: (1) The profiling approach is the most accurate, i.e. it surpasses the simulation approach, (2) it is faster than the simulation approach and thus more suited for the evaluation, and (3) it does not affect statements about the simulation approach if evaluated in favor of the hybrid approach. The third reason holds because the profiling is faster than the simulation, i.e. if the hybrid approach is faster than the profiling approach, it is faster than the simulation, and more accurate.

In order to evaluate these questions, we perform the hybrid dynamic code analysis on several different CNNs ($n = 32$) and compare these results to classic profilers. In the following, we describe the technical setup, used benchmarks, and computed metrics, before presenting the evaluation results.

Technical Setup: The technical setup is a SLURM-based HPC cluster and we ensure that the same machine of the cluster is used in all experiments. The used machine is equipped with three Nvidia V100S 32GB, 256GB memory, and 2 AMD EPYC ROME 7272. The home directory is a *Network Attached Storage* (NAS) connected by a 10GBit/s ethernet connection. HyPA runs on a Lenovo ThinkPad T490s with Intel i7-8565U, 16GB memory and Ubuntu 22.04.

Benchmark CNNs: The following provides an overview of the CNNs used for all experiments. They differ in various aspects like the number of layers, neurons, or input layer size. In Table I, the CNNs and their attributes are listed.

Moreover, we consider CNNs designed for different use cases. While some CNNs are designed to reach the best prediction accuracy, like Resnet [6], Alexnet [35], or Densenet [36], some are designed to perform well on mobile devices like MobileNet [37]. In contrast, the NASnetmobile and NASnetlarge are designed by *Neural Architecture Search* (NAS) [38] techniques.

All CNNs are pre-trained and downloaded from Tensorflow Hub as we focus on the inferencing aspect of these networks.

Data Generation: For each CNN, two profiles are computed: One, as a result of the application of HyPA, and one as a result of the application of nvprof as the reference value. In order to determine the average execution time of both HyPA and nvprof for the profile generation, each profile generation is repeated 10 times. The resulting metrics (besides run-time) are identical for all runs, as both approaches are deterministic.

Applications' Metrics: We measure the performance of nvprof and HyPA in collecting the following metrics: *branch_efficiency*, *inst_executed*, *flop_count_sp*, *flop_count_dp*, and *flop_count_hp*, where the latter three are cumulated to *flop_count*.

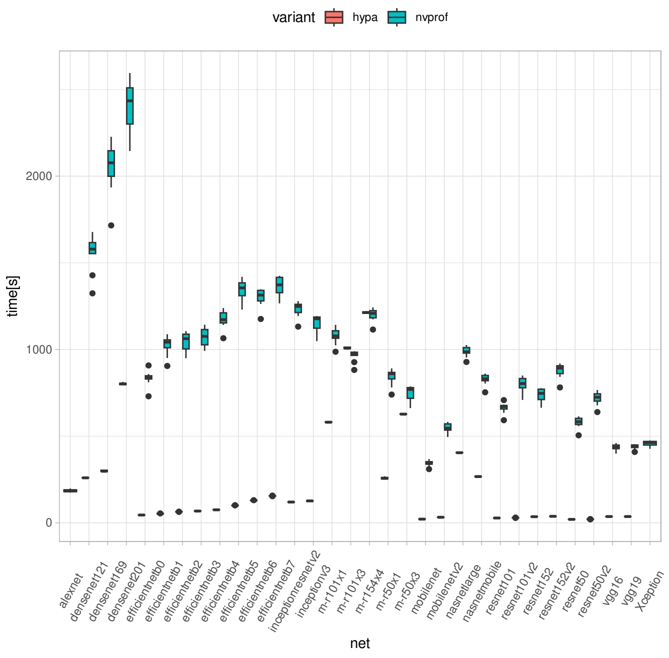

Fig. 3: Run-time comparison for HyPa and nvprof

The Branch Efficiency (η) is defined as the ratio between the number of branches (b) and divergent branches (d) and can be calculated as follows [39]:

$$\eta = 100 \times \frac{b - d}{b} = 100 - 100 \times \frac{d}{b} \tag{4}$$

If there are no divergent branches, the efficiency is 100%, if every branch is a divergent branch, the efficiency is 0%.

Experimental Results: Figure 3 illustrates the runtime results for both HyPA and nvprof as boxplot comparisons. For every net, the right boxplot shows the nvprof results, and the left boxplot shows the HyPA results. The results of HyPA are smaller and show less variance than the results of nvprof. This decreases the visibility of the boxplots' colors. Therefore, Figure 4 visualizes this relation again, but with focus on the pairwise comparison of results. The different colors indicate different CNNs, while the line depicts the bisector. Values above the bisector indicate greater runtimes for HyPA, values below the bisector show greater runtimes for nvprof. Most nets have shorter runtimes for HyPA than for nvprof. The results for the execution time yield mean values between $185.3s$ and $2410.1s$ for nvprof, mean values between $19.63s$ and $1220.1s$ for HyPA. For 27 out of 32 CNNs, the reduction in runtime is statistically significant ($\alpha < 0.05$, Welsh's t-test for comparing means).

Regarding the acquired metrics, our experiments are illustrated in Table II. It shows values for the instruction count results, counts of floating-point operations and results for branch efficiency. Additionally, we report the percentage of instructions that need execution (simulation) for HyPA. Results for both types of instructions show varying results. In most cases, HyPA reports a smaller instruction count than the static analysis which may be due to the conservative overestimation of static analyses. For seven CNNs, the HyPa reports a higher instruction count than static analysis. The instruction count results for nvprof are significantly higher than both HyPA and static analysis. We discuss this in the next section.

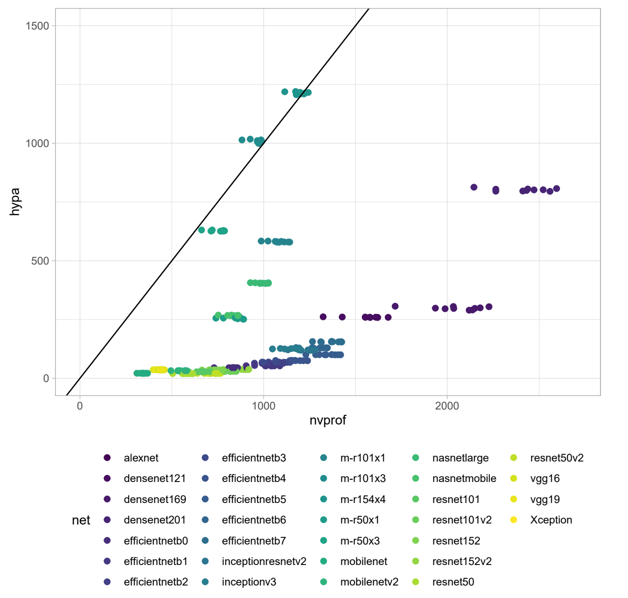

Fig. 4: Run-time comparison for HyPa and nvprof in correlation

As HyPA considers runtime information, it is capable of reporting metrics such as branch efficiency which is impossible for static analysis by design. The reported values of HyPA for branch efficiency deviate by 0.5 - 7 percentage points from the ones reported by nvprof, with one outlier that deviates by 12 percentage points.

Overall, HyPA had to perform at most 10% of the instructions to obtain the necessary information to compute its required metrics. This is in line with the runtime results described above.

V. Discussion

Following, we will discuss the interpretations of our evaluation and the current limitations of HyPA.

A. Evaluation interpretation

In response to R1, we found that HyPA is faster than nvprof in 27 out of 32 experiments. Therefore, it can be assumed that HyPA is also faster than classical simulation, such as GPGPU-Sim, since a simulation is slower than profilers like nvprof.

When analyzing R2, we found a significant difference between the calculated instructions by HyPA and the actual values measured by nvprof. This difference can be explained by three reasons:

1) PTX instructions are not directly translated one-to-one into SASS instructions, and since NVIDIA does not provide documentation on SASS, it is difficult to understand the translation process [18].
2) During the translation process, further optimization steps can be performed, resulting in one PTX instruction being translated into multiple SASS instructions [18].
3) CUDA kernels can be called and executed multiple times, which is not evident in the PTX code but can be observed in the nvprof profiles.

Comparing nvprof metrics like *instruction_executed* and *flops_count_sp* can be complicated because some work on wrap-level, while others work on thread-level [18].

Verifying points 1 and 2 through reverse engineering is complicated. However, verifying point 3 is possible by simulating parts of the CUDA code to obtain the exact number of kernel calls. Therefore, integrating HyPA into a compiler such as LLVM or Clang can help with this step. It is essential to apply the concept of HyPA to both CUDA and PTX, not just PTX.

Finally, we can answer R3 and demonstrate that our system can gather metrics beyond what static code analysis can provide, such as branch efficiency. Additionally, we have found that our system can accurately predict branch efficiency with a high degree of accuracy, ranging from 0.5 to 7 percentage points off from what is reported by nvprof. When combined with the speedup, our the system represents a critical improvement for power and performance estimation, as well as the extraction of necessary metrics.

B. Limitations

Besides all its merits, HyPA still has some drawbacks and limitations. Currently, two aspects limit our approach 1) not all PTX instructions are implemented, and 2) data dependencies in conditional jumps might not resolve correctly. In the following, we describe each limitation and a possible solution to overcome these.

Since not all PTX instructions are implemented (e.g., indirect addressing), our HyPA implementation has to fall back on state-of-the-art approaches. Thus, the results might still be approximated. In future work, we plan to implement an equivalent for all PTX instruction and, thus, overcome this issue.

Regarding the second limitation: In some cases, a register is written in two parallel conditional jumps. Currently, HyPA is adding an edge to the first node in the dependency graph where a source register is written. This might lead to a wrong control flow when the register is written in both branches. This issue can be solved by implementing an *Static Single Assignment* (SSA) Form y [40], [41] or by earlier dynamic analysis and determination of which branch is to be taken into account. Thus, the edge for the dependency can be added between the correct nodes instead of the earliest occurring node that writes to a register and thus generates a data dependency. We plan to modify the dependency graph generation and extend it by control flow generation with SSA Form to overcome this issue and improve HyPA to get closer to the most accurate calculation of the total number of instructions.

VI. Related Work

The performance evaluation and optimization process can be divided into two categories 1) given a particular hardware, the software is optimized to extract the maximum performance on the architecture, and 2) given a group of applications (software), hardware features can be extracted to adapt novel software requirements [20].

In order to perform optimization analysis, tools (profiler) are necessary. Most (GP)GPU vendors provide tools like nsight, CUPTI, nvprof [15], or ncu from NVIDIA. Moreover, alternative tools for multi-vendor applications are more comprehensive than just a single vendor or system. [20]. However, using profilers and performance counters is limited and has three main disadvantages 1) the number and type of performance counters are not uniform over different devices (e.g., GPGPUs) even if the same profiler is used. Hence, performance and power optimization techniques or predictive models based on performance counter are not immediately generalizable. 2) The number of performance counters that are simultaneously accessible is limited by the number of hardware registers. Consequently, multiple application executions and profiling may be required to collect all performance counters [17]. This leads to longer profiling times and limits the estimation of profiling run-time. 3) To measure the different performance counters, the application must be run on actual GPUs. In the case of searching for the most appropriate GPU,

TABLE II: Experimental results for static and dynamic analysis with HyPA for 32 different CNNs.

	NVIDIA V100s									
		Instruction Count						Branch Efficiency		
CNN	**% to execute**	**HyPA**	**Static**	**ratio**	**nvprof Instruction**	**nvprof FLOPs**	**HyPA FLOPs**	η_{nvprof}	η_{HyPA}	ratio
densenet121	9.46%	3,347,534	10,722,398	0.3122	73,918,277,390	1,239,313,864,022	340,755	92.1568%	98.9158%	0.9316
densenet169	9.76%	4,618,062	19,331,166	0.2388	132,677,813,819	2,323,797,491,957	392,979	92.1568%	99.0541%	0.9303
densenet201	9.86%	6,975,950	26,674,398	0.2615	268,813,644,433	4,853,604,008,393	478,995	92.1568%	99.1078%	0.9298
efficientnetb0	7.21%	1,605,049	3,563,417	0.4504	28,170,395,486	341,116,792,220	356,583	92.1535%	96.0827%	0.9591
efficientnetb1	6.24%	2,284,762	4,870,436	0.4691	44,518,197,354	617,624,476,202	568,808	92.1535%	96.2471%	0.9574
efficientnetb2	5.76%	3,234,232	6,097,016	0.5304	74,094,157,251	824,407,223,138	776,827	92.1535%	96.3291%	0.9566
efficientnetb3	6.40%	3,785,200	6,140,976	0.6163	110,536,344,626	1,103,881,214,923	894,237	92.1535%	96.8977%	0.9510
efficientnetb4	6.04%	4,167,634	6,454,880	0.6456	128,613,616,058	1,502,321,376,491	1,014,889	92.1535%	97.2130%	0.9479
efficientnetb5	5.57%	6,988,730	10,670,869	0.6549	154,166,225,231	1,999,289,837,215	1,691,145	92.1535%	97.5638%	0.9445
efficientnetb6	6.13%	8,311,211	12,591,814	0.6600	133,278,295,936	2,212,260,957,075	2,037,615	92.1535%	97.9676%	0.9406
efficientnetb7	6.28%	9,979,998	13,901,223	0.7179	98,031,779,822	2,251,834,391,082	2,362,090	92.1535%	97.6439%	0.9437
InceptionResNetV2	7.89%	4,986,406	9,935,678	0.5018	128,538,060,350	2,591,083,345,663	666,852	92.1568%	99.1184%	0.9297
inceptionv3	7.06%	3,591,286	6,674,334	0.5380	99,139,487,866	1,709,094,946,347	429,036	92.1568%	98.6580%	0.9341
mobilenet	10.77%	792,566	1,077,652	0.7354	15,516,090,366	287,752,628,102	169,807	92.1086%	97.3356%	0.9462
MobileNetV2	10.22%	995,962	2,032,606	0.4899	15,757,543,754	210,300,655,245	177,403	92.1538%	98.2702%	0.9377
m-r101x1	7.11%	96,990,375	28,257,288	3.4324	45,603,141,589	870,125,486,565	10,790,977	92.1568%	98.6123%	0.9345
m-r101x3	8.43%	187,338,305	42,787,720	4.3783	101,530,724,333	2,734,139,342,013	21,533,361	99.6015%	99.1282%	1.0047
m-r154x4	9.53%	267,066,941	62,232,168	4.2914	156,582,406,489	4,138,138,061,309	34,877,039	99.6015%	99.2344%	1.0036
m-r50x1	8.28%	46,505,966	14,963,624	3.1079	44,031,032,572	853,827,859,109	5,338,521	92.1568%	98.5723%	0.9349
m-r50x3	9.02%	90,945,907	20,466,728	4.4435	91,364,961,407	2,589,336,187,773	10,241,279	99.6010%	99.0358%	1.0057
nasnetlarge	7.55%	5,776,912	28,546,814	0.2023	76,928,601,211	1,702,504,253,055	568,407	92.1566%	99.2786%	0.9282
nasnetmobile	8.31%	3,363,489	19,124,047	0.1758	8,651,631,107	103,056,406,909	192,899	92.1536%	99.1792%	0.9291
resnet101	5.77%	2,728,871	2,922,782	0.9336	33,718,778,616	665,107,865,206	625,983	92.1565%	98.8248%	0.9325
resnet101v2	6.13%	2,804,054	3,035,678	0.9236	53,668,419,497	1,109,393,289,019	570,448	92.1565%	98.8290%	0.9324
resnet152	5.50%	3,923,239	4,117,022	0.9529	34,822,490,986	680,214,892,914	917,567	92.1565%	99.0168%	0.9307
resnet152v2	5.83%	4,012,758	4,248,094	0.9446	54,769,117,374	1,124,161,761,326	864,080	92.1565%	99.0196%	0.9306
resnet50	6.51%	1,532,071	1,725,982	0.8876	32,615,069,876	650,429,295,715	334,399	92.1565%	98.2734%	0.9377
resnet50v2	6.88%	1,605,974	1,832,990	0.9761	52,567,694,245	1,094,624,830,407	280,912	92.1565%	98.2815%	0.9376
vgg16	9.56%	8,738,852	412,254	21.1977	96,577,294,513	1,480,562,789,621	1,292,884	92.0827%	86.9564%	1.0589
vgg19	9.56%	8,738,852	412,254	21.1977	94,645,096,385	1,488,245,837,048	1,292,884	92.0827%	86.9564%	1.0589
Xception	6.37%	2,223,811	3,402,190	0.6536	24,125,255,065	735,441,505,983	391,266	76.3605%	98.4047%	0.7759

many different GPUs are needed, and the profiling process must be repeated on each of them. This is a very time-consuming and costly process.

CUDA Flux [18] is a lightweight Instruction Profiler for CUDA applications that characterizes all types of PTX instructions executed by a kernel. However, CUDA Flux is not considering divergent branches within a PTX code. Hence, it cannot count the actual number of executed instructions and is still approximating. In contrast, our approach analyses the PTX code by combining static analysis with partial simulation. However, CUDA Flux is only a static analysis of the PTX code. Moreover, the accuracy is only given for single Thread analysis.

Besides the analysis tools, there is also a set of simulation-based approaches like GPGPU-Sim [21], ocelot [22], or Barra [24]. The obtained results have an accuracy between 10% to 20% compared to the actual hardware execution [42]. However, these simulators execute the GPU code on the CPU, which does not offer a high amount of parallelism and, thus, is significantly slower than the actual hardware execution. The limited accuracy and speed are significant drawback that impairs the usage of simulators for GPU application profiling [20]. Moreover, GPGPU-Sim is more suitable for evaluating GPU architecture and helps designers modify existing GPU architectures by adding detailed timing models [23]. In contrast, we focus on analyzing high-level PTX kernels' insights.

In [12], a *Virtual GPU* (VGPU) is introduced. The OpenMP code, specified to run on a GPU, is compiled to CPU binary but includes the same instruction as for the GPU. This allows the execution of the GPU code on a CPU while using the available CPU profiling tools. However, since the highly parallel GPU code is executed on the CPU, it has the same disadvantages as the simulator. Moreover, the work focuses on GPU applications written in OpenMP to make the application portable between the different GPU vendors like NVIDIA and AMD. In contrast, we focus on CUDA applications since CUDA is the most common GPU implementation for neural network frameworks.

Similar to the simulators are *Virtual Prototypes* (VPs) [43], which were the inspiration for the execution implementation of different PTX instructions on the CPU in C++. A VP is an abstract model of a specific hardware platform (e.g., RISC-V processors). Using a VP enables software developers to simulate applications behavior at early design stages on the virtual device.

In [20], a top-down performance profiling approach for NVIDIA's GPUs is presented. Therefore, they take advantage of the profiling tools offered by NVIDIA (e.g., CUPTI, nvprof, and ncu). Consequently, the application has to be executed on an actual device. In order to verify multiple GPUs' performance, the top-down performance profiling from [20] has to be executed on multiple actual GPUs.

VII. Conlusion

We created a tool that can analyze PTX code without having to run it on a real GPGPU. To account for divergent branches, we use partial execution and code emulation. By reimplementing the PTX ISA in C++, we can emulate the execution on a CPU system. Unlike traditional GPU simulators that run the entire application, our approach only executes the necessary parts to identify divergent branches. This speeds up the analysis, and the execution time is comparable to or faster than profiling with nvprof on the NVIDIA V100S.

Our research shows that HyPA can provide metrics beyond what static code analysis can offer, such as branch efficiency. One significant advantage of HyPA is its ability to speed up the design space exploration process by providing early metrics for different approaches to power and performance prediction of CNNs on GPGPUs. HyPA's quick execution time makes it an excellent tool for this type of work.

Although HyPA is capable of parsing the entire PTX ISA, it currently has a limitation in its ability to emulate indirect addressing.

As part of our future work, we aim to implement indirect addressing in order to fully cover the ISA emulation. We will expand the hybrid analysis to include memory access emulation for analyzing memory reads and writes. This will result in a more comprehensive profile that is better suited for deep learning application profiling.

References

[1] J. Tang, D. Sun, S. Liu, and J.-L. Gaudiot, "Enabling deep learning on iot devices," *Computer*, vol. 50, no. 10, pp. 92–96, 2017.

[2] F. Foertter, "Summit gpu supercomputer enables smarter science," https://developer.nvidia.com/blog/summit-gpu-supercomputer-enables-smarter-science/, 2018, accessed: 22.03.2022.

[3] TOP 500, "List statistics," https://www.top500.org/statistics/list/, 2022.

[4] M. Feldman, "New gpu-accelerated supercomputers change the balance of power on the top500," https://tinyurl.com/2p8x74ww, 2018.

[5] Nvidia, "Nvidia v100 tensor core gpu," https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf, 2020, accessed: 2022-11-02.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[8] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: training imagenet in 1 hour," *CoRR*, vol. abs/1706.02677, 2017.

[9] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely large minibatch SGD: training resnet-50 on imagenet in 15 minutes," *CoRR*, vol. abs/1711.04325, 2017.

[10] J. Guerreiro, A. Ilic, N. Roma, and P. Tomás, "Gpu static modeling using ptx and deep structured learning," *IEEE Access*, vol. 7, pp. 159 150–159 161, 2019.

[11] J. Doerfert, A. Patel, J. Huber, S. Tian, J. M. M. Diaz, B. Chapman, and G. Georgakoudis, "Co-designing an openmp gpu runtime and optimizations for near-zero overhead execution," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 504–514.

[12] A. Patel, S. Tian, J. Doerfert, and B. Chapman, "A virtual gpu as developer-friendly openmp offload target," in *50th International Conference on Parallel Processing Workshop*, ser. ICPP Workshops '21. New York, NY, USA: Association for Computing Machinery, 2021.

[13] C. A. Metz, M. Goli, and R. Drechsler, "Ml-based power estimation of convolutional neural networks on gpgus," in *IEEE International Symposium on Design and Diagnstics of Electronic Circuits and Systems*, 2022, pp. 166–171.

[14] ——, "Towards neural hardware search: Power estimation of cnns for gpgpus with dynamic frequency scaling," in *Proceedings of the 2022 ACM/IEEE Workshop on Machine Learning for CAD*, ser. MLCAD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 103–109.

[15] Nvidia, "Profiler user's guide," https://docs.nvidia.com/cuda/profiler-users-guide/index.html, accessed: 2022-02-02.

[16] NVIDIA, "Cupti documentation," https://docs.nvidia.com/cupti/index.html, 2022.

[17] R. A. Bridges, N. Imam, and T. M. Mintz, "Understanding gpu power: A survey of profiling, modeling, and simulation methods," *ACM Comput. Surv.*, vol. 49, no. 3, sep 2016.

[18] L. Braun and H. Fröning, "Cuda flux: A lightweight instruction profiler for cuda applications," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019, pp. 73–81.

[19] D. Lustig, S. Sahasrabuddhe, and O. Giroux, "A formal analysis of the nvidia ptx memory consistency model," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 257–270.

[20] A. Saiz, P. Prieto, P. Abad, J. A. Gregorio, and V. Puente, "Top-down performance profiling on nvidia's gpus," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 179–189.

[21] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174.

[22] gatech, "GPU ocelot," https://gpuocelot.gatech.edu/doxygen/, 2012.

[23] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of ptx kernels," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 3–12.

[24] C. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A parallel functional simulator for gpgpu," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010, pp. 351–360.

[25] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, p. 152–163, 2009.

[26] P. Busia, S. Minakova, T. Stefanov, L. Raffo, and P. Meloni, "Aloha: A unified platform-aware evaluation method for cnns execution on heterogeneous systems at the edge," *IEEE Access*, vol. 9, pp. 133 289–133 308, 2021.

[27] Nvidia, "Volta architecture whitepaper," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, accessed: 2022-01-18.

[28] NVIDIA, "Kernel profiling guide - user manual," https://docs.nvidia.com/nsight-compute/2020.1/pdf/ProfilingGuide.pdf, 2020.

[29] Nvidia, "CUDA toolkit documentation," https://docs.nvidia.com/cuda/, 2022.

[30] ——, "Parallel thread execution isa," https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[31] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of gpu architectures," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 185–197.

[32] S. Hong and H. Kim, "An integrated gpu power and performance model," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 280–289, 2010.

[33] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1981, pp. 207–218.

[34] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *ACM SIGPlan Notices*, vol. 25, no. 6, pp. 246–256, 1990.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012.

[36] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks."

[37] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017.

[38] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *CoRR*, vol. abs/1707.07012, 2017.

[39] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA c programming*. John Wiley & Sons, 2014.

[40] A. W. Appel, "Ssa is functional programming," *Acm Sigplan Notices*, vol. 33, no. 4, pp. 17–20, 1998.

[41] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989, pp. 25–35.

[42] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 564–576.

[43] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *JSA*, vol. 109, p. 101756, 2020.