Characterization of Artificial Intelligence Accelerators for Timing Analysis

Fachhochschule Dortmund

Master Thesis

submitted by
Ali A. K. Alhalabi
Embedded Systems Engineering
Mat.Nr.: 7216390
ali.akalhalabi001@stud.fh-dortmund.de

May 12, 2025

First supervisor: Prof. Dr. Stefan Henkler
Second supervisor: Prof. Dr. Martin Hirsch

Contents

Abstract - Estimating the worst-case execution time (WCET) of an artificial intelligence (AI) inference operation has become more important recently as AI is increasingly being deployed using specialized accelerators in various applications, including time critical applications. Current approaches tend to focus on one execution paradigm and attempt to correlate it with Worst Case Execution Time failing to take into account the impact of other factors and features. This work deals independently with two types of accelerators, Systolic-based accelerators that can perform a small subset of AI-relevant operations, and Nvidia GPUs that provide thousands of parallel cores that perform the same operation on different data.

In this thesis, the study and analysis of systolic data flow is delved into in order to determine the number of multiply-accumulate (MAC) operations required to execute an AI-related matrix multiplication operation. For Nvidia GPUs, a comprehensive algorithm was developed that incorporates instruction delays, instruction level parallelism, thread level parallelism, and memory and cache behavior to estimate the number of clock cycles needed for a single GPU operation.

The dataflow analysis of systolic arrays is compared against other approaches and contemporary research, whereas the algorithm developed for Nvidia GPUs has been tested by statically analyzing the assembly instructions of the code based on the developed algorithm, and comparing it to the average runtime cycles reported by Nvidia-based analysis tools. The GPU used was Nvidia A100, however, the algorithm is designed in way to be scalable for most Nvidia GPUs for future work.

Keywords - Nvidia Graphics Processing Unit, Systolic Arrays, Artificial Intelligence Accelerators, Worst Case Execution Time, Compute Unified Device Architecture.

1 Introduction

The exploration of machine intelligence has been a prominent area of research since the advent of digital computers. Alan Turing, in 1950, introduced the "imitation game" [1], more commonly referred to as the Turing Test, to empirically explore whether "machines can think." In addition to this, Turing delved into the concept of "learning machines," which are capable of modifying their pre-programmed instructions based on their experiences. Following the introduction of the imitation game, advancements in Machine Learning (ML) and Artificial Intelligence (AI) have swiftly expanded, transforming numerous facets of everyday life. This has led to a significant shift in academic interest toward machine learning applications in the automotive industry and self-driving vehicles, healthcare, big data analytics, and more [2].

AI and ML are increasingly being utilized in applications that are both time-sensitive and privacy-conscious, presenting new challenges and requirements in the field. A notable development is the emergence of a new category of hardware capable of executing Artificial Intelligence locally on devices or at the edge, as opposed to relying on traditional cloud-based machine learning models. This shift is equipping real-time systems with the ability to run ML and AI models using hardware accelerators like Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and other Application-Specific Integrated Circuits (ASICs) specifically optimized for AI-related computations. A diverse array of hardware, developed by various manufacturers, is now available for AI and ML applications, shown in Figure 1.1. These range from low to high power consumption and include specific features for purposes like training, inference, or both. This includes GPUs capable of, or specifically designed for, running AI and ML models, such as those produced by Nvidia, or ASICs like the Google Tensor Processing Unit (TPU), built specifically to execute certain AI models. The Intel Movidius Neural Compute Stick, designed to run AI models on the edge, exemplifies the current trend and importance of executing AI and ML models in real-time and on the edge.

AI accelerators are specifically designed and optimized to work with Deep Neural Networks (DNNs), a subset of Artificial Neural Networks (ANNs), which structurally emulate the human brain's neural cells. These DNNs find applications across various domains, including image and speech recognition and natural language processing. Composed of multiple interconnected layers of neurons, DNNs are trained using extensive data sets to discern patterns and features through weighted significance. The training phase is computationally intensive, consuming substantial amount of data and power. Once trained, the model is employed for inference, either locally or in the cloud, where it predicts new data outcomes. This inference process demands fewer computational resources than training and enables real-time applications [4, 5]. The proliferation of domain-specific computing devices, like AI accelerators, emerges from the necessity to execute DNNs in real-time and on edge devices [5]. These AI accelerators consist of highly parallel processing units crafted to perform

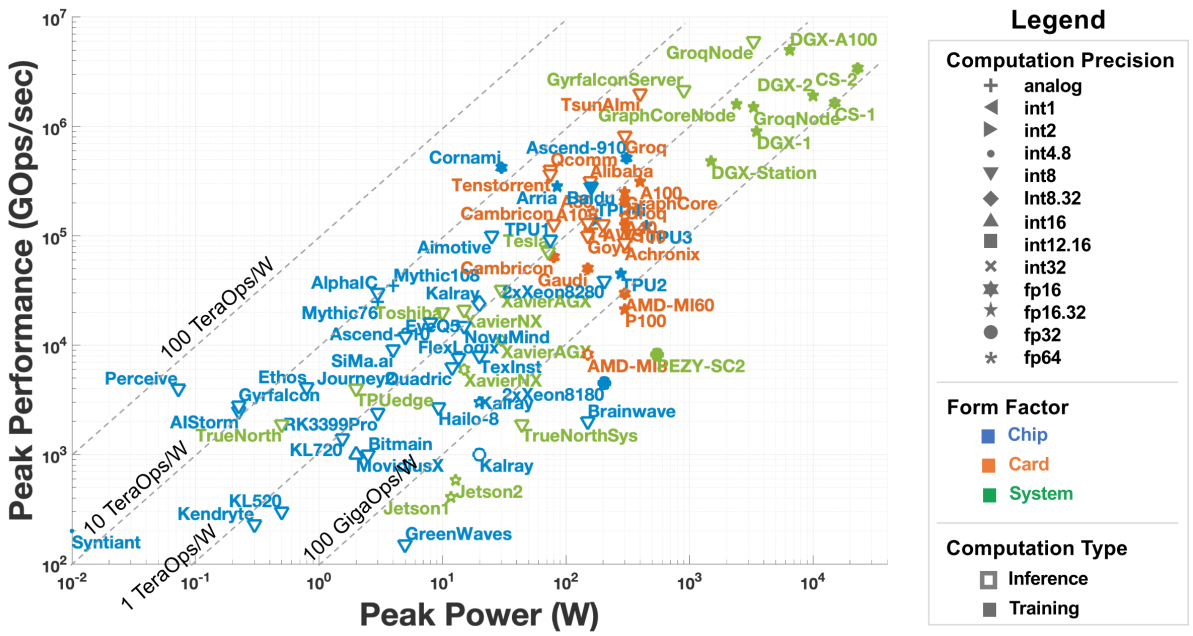


Figure 1.1: Trends of AI Accelerators usage based on power, performance, precision, and task [3].

large-scale matrix multiplications, a pivotal operation in DNNs for computing neuron outputs across network layers, with matrix multiplication and convolution accounting for 90% of used operations [5].

Incorporating these devices into real-time systems introduces regulatory and reliability challenges, where execution time must be validated and tested in advance. This validation is crucial for ensuring reliability and predictability, necessitating a profound understanding of hardware and software interactions. A vital metric in this context is the Worst-Case Execution Time (WCET), representing the maximum duration a task requires on a given hardware platform. However, computing the WCET for GPUs and other AI accelerators differs significantly from that of single or multicore Central Processing Unit (CPU) systems, due to the hardware's highly parallel nature, lack of Instruction Set Architecture (ISA) documentation, complex memory hierarchies, and manufacturers' focus on enhancing average-case performance [6, 7, 8].

In the following Section 1.1, a motivation for the thesis topic will be presented and an overview of the goals of this thesis is provided in Section 1.2.

1.1 Motivation

Edge and on-device AI, have witnessed a huge attraction in recent years as a complement to cloud computing, where latency is minimized, and high bandwidth is not required [9]. This can also be seen in the trend of AI accelerators that perform inference on the edge with low power requirements, shown in Figure 1.1. However, with the rise of AI accelerators, specifically in the case of GPUs, the focus has been on improving the average-case performance and not on the worst-case performance or predictability [10], which impacts the use cases in which AI on the edge can be used.

Liang et al. [9] conclude that edge computing with accelerators renders better normalized performance than cloud computing, provides support for various degrees of concurrency, but lacks the isolation provided in cloud computing. However, in safety-critical systems, such as autonomous vehicles, medical devices, and industrial control systems, etc., more accurate metrics of execution are required. The reliability and predictability of a system is a requirement that needs to be met, tested, and guaranteed beforehand. This is where WCET analysis plays a crucial role as it is a key metric for real-time systems and identifies key factors that affect the execution of a task [11].

Betts et al. [11] acknowledge the maturity level of WCET analysis for single-core uniprocessor systems, but state that the same cannot be said for GPU-based systems or other parallel systems due to the limitations of static analysis methods that depend on the ISA, the hardware architecture, and the complexity of the single instruction, multiple threads (SIMT) architecture used in GPUs.

This research seeks to address the gap identified by Betts et al. [11] and others regarding the application of Worst-Case Execution Time (WCET) analysis to AI-accelerated systems. Current research trends focus on analyzing only one of the multiple execution paradigms, neglecting to integrate with other approaches for different execution paradigms. Additionally, there's a gap in the application of static analysis methods due to the absence of a usable execution model. This research gap, stemming from the lack of a practical execution model and a deeper analysis, is the driving force behind this thesis, which aims to address these issues and define the research questions in the next chapters.

1.2 Goals

The integration of AI models into edge and embedded devices prompts some additional requirements regarding timing correctness. This research seeks to address the primary research question: **What is the worst-case execution time (WCET) of an inference operation executed on an AI accelerator?** To achieve this, the question is divided into three distinct research questions:

1. Can WCET be applied to AI Accelerators?
2. What operations exist in AI inference?
3. How do AI accelerators execute these operations?

There exists a plethora of research with respect to the first two questions, however, for the third question, the research gap grows as it depends on the execution model of the accelerator. In order to answer the main research question, more information regarding the execution model of the accelerators needs to be studied. The work in this thesis focuses on identifying and characterizing the differences between AI accelerators to understand their execution model, i.e. how do they execute instructions? This will be proven extremely necessary for WCET analysis. This thesis makes the following contributions to address the research questions:

- In-depth analysis and characterization of AI Accelerators.
- Modeling the Execution cycles of Systolic Accelerators.
- Modeling the Execution cycles of Nvidia GPUs.
- Identifying Cache and memory behavior of Nvidia GPUs.
- An algorithm for estimating WCET of Nvidia GPUs executables.

These findings aim to support the timing predictability of AI models in real-time and safety-critical applications. WCET is a fundamental requirement for the design and validation process of such applications, since missing a deadline can lead to system failure or safety concerns.

2 Fundamentals

The Fundamentals chapter aims to answer the first two research questions: whether WCET can be applied to AI Accelerators and what operations exist in AI inference. Worst-case execution time (WCET) is covered in Section 2.1 as a starting point for this research. The fundamental concept of AI is discussed in Section 2.2, which aims to demystify some preconceptions behind AI as a concept. Some of the pillars of AI operations are explored in Section 2.3, which delves into Multi-layer perceptrons, convolution neural networks, and Transformers. Additional information on the general execution of these operations is provided in Section 2.4.

2.1 Worst-Case Execution Time

Worst-Case Execution Time (WCET) is typically associated with real-time systems, which need to deliver logically correct results under a specific, strict, or semi-strict deadline that matches the environment they operate in [12]. AI is becoming more and more involved in less obscure and niche use-cases and is gaining traction in applications like automated driving, among others. Real-time systems react to stimuli from the environment within a given time frame relevant to that environment, called a deadline, and there would be consequences if this deadline is missed [12].

Worst-Case Execution Time (WCET) analysis involves determining an upper limit for an executable's[1] execution duration. The WCET bound is influenced by the input data space, code logic, and the characteristics of the target hardware [13]. Appropriate WCET analysis should produce safe estimation that covers real execution scenarios and avoids infeasible execution scenarios. The description of execution-time cannot be described in one straight-forward term, but as a set of metrics that describe the execution of an executable. Figure 2.1 shows a set of metrics related to execution time and are described in Table 2.1

Figure 2.1 Shows that the measured execution time is less than the WCET. This means that WCET requires some structured approach to obtain, rather than measuring the execution time as WCET serves as an upper execution bound that is guaranteed not to be exceeded [12]. In order to do so, an executable is analyzed under strict, realistic input domain that matches the environment where all execution paths are explored. But in general, a typical static WCET approach depends on the representation level of the executable, execution model of the hardware, and the flow facts of the executable, these aspects are explained in [13] as the following:

[1] Any piece of code that runs on hardware, including full programs, specific sections of code, individual functions, etc.

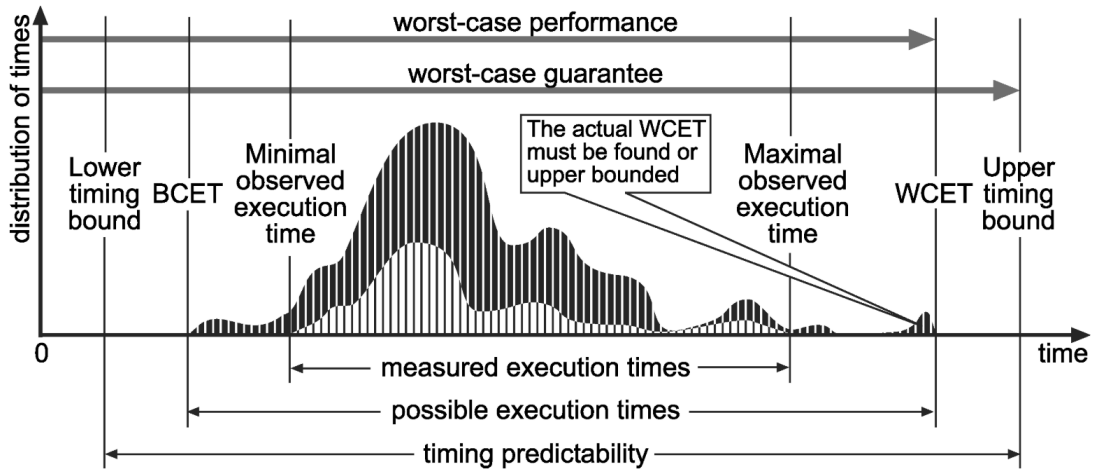


Figure 2.1: Timing analysis of an executable [14]: The distribution of execution time of an executable.

- **Representation Level:** Describes which level of code on which the WCET analysis is performed, high-level languages are more complicated to analyze and produce less accurate WCET compared to analysis done on lower-level representations such as assembly or object-code which produce an accurate WCET, typically referred to as tight WCET. Even though programming in languages such as assembly is discouraged, most modern compilers are capable of providing an assembly representation of the code during the compilation process. It is important to mention that compilers tend to change the structure of code for optimization purposes, producing different execution paths compared to the higher-level representation.

- **Execution Model:** Models the hardware that executes the produced piece of code. WCET is directly dependent on the execution model of the hardware, including delays of instructions, paths and branch prediction, memory and cache behavior, pipelines, etc. The more accurate the execution model, the tighter the WCET would be. However, it is unfeasible to have or produce an exact execution model, which might complicate the WCET analysis further, and it is typically sufficient to find a compromise between WCET accuracy and complexity of analysis.

- **Flow Facts:** Describes constraints on the execution of a program, flow facts can be inferred from inspecting the source code of the program and the kind of operation a code performs, or through formal methods such as abstract interpretation, or symbolic execution, both of which are methods of static analysis that explore execution paths [15]. Similarly to representation level, compilers tend to optimize and modify the flow of the executable beyond what is programmed in the original source code.

A typical approach to WCET analysis depends on the source code as an input, the source code could be in a high-level language such as C, C++, Java, etc. or lower level representation such as binary or assembly. The flow facts can then be extracted from the source to be used to restrict the execution of the code, and to be adjusted throughout the compilation process. Then, after compilation, the object code is to be analyzed against the execution model to produce the WCET estimation [13]. The process of WCET analysis

Term	Description
Worst-case Execution Time (WCET)	The actual longest execution time of a task or a set of tasks. Usually cannot be directly measured due to large size of state space and variables affecting the system.
Best-Case Execution Time (BCET)	The actual shortest execution time of a task or a set of tasks. Usually cannot be directly measured due to large size of state space and variables affecting the system.
Minimal observed execution time	The shortest measured end-to-end execution time of the task for a subset of the possible executions with test cases - This generally overestimates the BCET.
Maximal observed execution time	The longest measured end-to-end execution time of the task for a subset of the possible executions with test cases - This generally underestimates the WCET.
Lower execution bound	The shortest execution time obtained by methods that abstract the task. Tends to underestimate the BCET.
Upper execution bound	The longest execution time obtained by methods that abstract the task. Tends to overestimate the WCET.

Table 2.1: Glossary of timing analysis terms used in Figure 2.1 [14].

cannot be generalized and solved to any piece of code running on any system as this is equivalent the halting problem [12, 13] and thus each WCET problem requires a great deal of human intervention to introduce restrictions and flow facts to the system.

The WCET of an executable is not independent of the collective system where multiple different codes or tasks need to be executed. Therefore, the scheduling algorithm employed needs to be taken into account for a full view of the system. Schedulers are typically categorized into static (offline) schedulers that make scheduling decisions at compile time, or dynamic (online) that schedule during the runtime of the system [12]. Analysis related to scheduling will be discussed in Chapter 4.

The following Section 2.2 gives an overview of what an AI model is, and the operations used in AI models are explored in depth in Section 2.3 and Section 2.4 before exploring the state of the art of WCET analysis for AI accelerators in Chapter 3

2.2 Artificial Intelligence Overview

Multiple definitions of Artificial Intelligence (AI) exist, but they all revolve around the idea of thinking rationally and "humanly" [16]. Neuroscience and understanding the human brain, or how humans think, is critical to AI. Currently, scientists understand the importance of neurons (nerve cells) in the brain and realize the networks they form in order to perform some specific function. Figure 2.2 shows the structure of a nerve cell which has a body, dendrites branching out of the body, and a long fiber called the axon which stretches long to connect to other neurons (up to 1 meter) to form synapses. Signals across neurons are propagated as electrochemical reactions. Neurons form from 10 to 100,000 synapses with other neurons and can reform structures and connections based on

the signals they receive. [16]

In 1943, Walter Pitts and Warren McCulloch proposed a model of a neuron that could be simulated using a computer, as advancements in digital computer hardware technology and programming languages made this possible. This model is known as the McCulloch-Pitts model, and it is based on the idea that a neuron can be represented as a binary threshold unit that processes multiple inputs and produces a single output. The output is produced based on the sum of the inputs multiplied by the weights of the connections between the inputs and the neuron. The output is then passed through an activation function to produce the output of the neuron. [16]

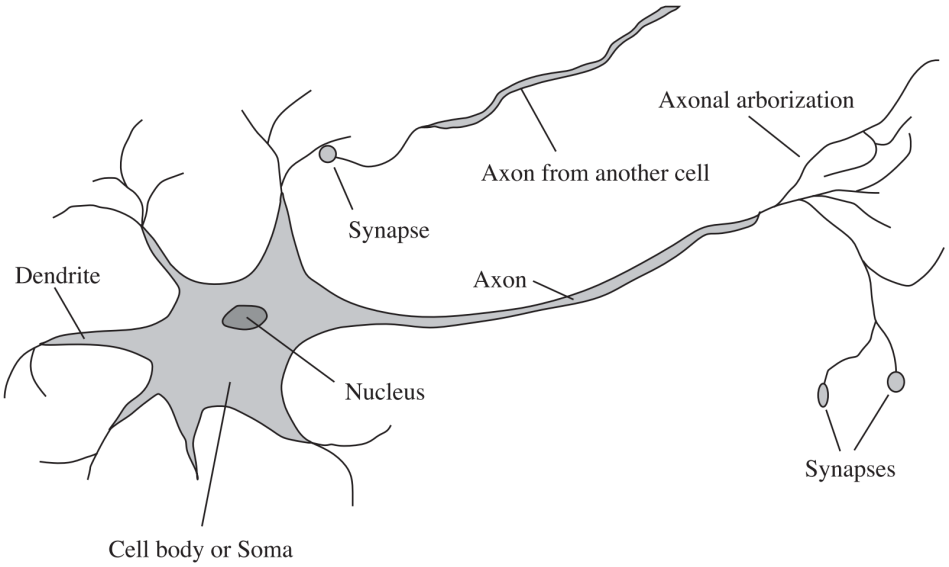


Figure 2.2: A biological nerve cell (Neuron), the Synapse of a cell connects to the Axon of other cells and propagate electrical signals [16].

A Neural Network (NN), including a simple multilayer perceptron, can be described as a parameterized function that infers an output, usually a pre-trained label, based on multidimensional input, usually expressed as a matrix [5]. Deep Neural Networks (DNNs) are considered the dominating method to deal with challenging machine learning applications [4]. Multiple forms of DNNs exist to solve different kinds of problems, such as Convolutional Neural Networks (CNNs) for image recognition and Recurrent Neural Networks (RNNs) for sequence data [17].

$$f(x) = f_{l-1} \circ f_{l-2} \circ \cdots \circ f_2 \circ f_1(x) \tag{2.1}$$

The structure of a DNN operation is typically represented in Equation 2.1 where f_i is the i^{th} layer of the network and x is the input to the network. The output of the network is the output of the last layer f_{l-1} [5]. A simple way to think of DNNs is to reduce a whole DNN model to a multi-layer perceptron (MLP) where a set of inputs on the first layer are processed by the value of these inputs multiplied by the weights assigned between the connection between the neurons plus the bias of each neuron of the second layer, which then is passed through an activation function to produce the output of the neuron. The first layer is known as an Input layer, the last layer is known as the output layer, and the

layers in between are known as hidden layers where feature extraction happens based on setting the weights and biases accordingly. The output of the last layer is the output of the network as a level of accuracy of a certain pre-trained label.

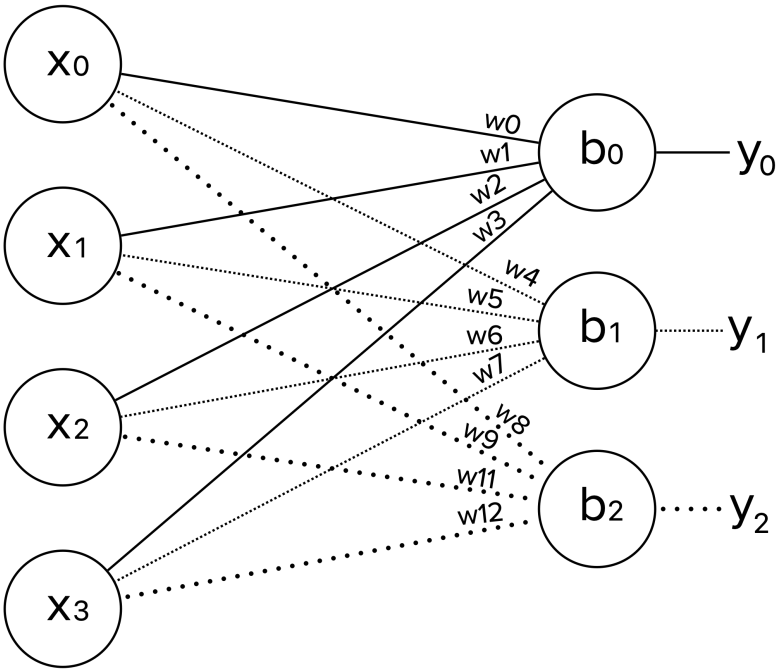


Figure 2.3: One layer of an MLP: $x_{0...3}$: inputs, $w_{0...12}$: weights, $b_{0...2}$: biases, $y_{0...2}$: outputs/inputs to next layer.

$$y0 = (x0 \cdot w0) + (x1 \cdot w1) + (x2 \cdot w2) + (x3 \cdot w3) + b0 \tag{2.2}$$

$$y1 = (x0 \cdot w4) + (x1 \cdot w5) + (x2 \cdot w6) + (x3 \cdot w7) + b1 \tag{2.3}$$

$$y2 = (x0 \cdot w8) + (x1 \cdot w9) + (x2 \cdot w10) + (x3 \cdot w11) + b2 \tag{2.4}$$

Figure 2.3 shows an example of one layer of an MLP, showing the input, labeled as X_i replacing x in Equation 2.1, connected to a hidden layer that has 3 neurons each labeled as B_i representing $f_1(x)$ in Equation 2.1, the input layer has no biases, but the hidden layer has biases represented by the assigned value of B_i. The weights are represented by W_i. The output of each neuron in this layer is calculated in Equations 2.2, 2.3, and 2.4. Usually, an activation function is applied to the output of each neuron such as Rectified Linear Unit (ReLU) function and the Sigmoid functions shown in Equation 2.5 and Equation 2.6 respectively.

$$f(y_i) = max(0, y_i) \tag{2.5}$$

$$f(y_i) = \frac{1}{1 + e^{-y_i}} \tag{2.6}$$

It is generally more inefficient to evaluate the output of each neuron individually. A more compact way to represent Equation 2.2, Equation 2.3, and Equation 2.4 is to represent them in matrix form as shown in Equation 2.7, where W represents a matrix of the weights, X represents the input vector from one layer to the other (commonly referred to as activations), and B represents the bias of each neuron [2]. Equation 2.7 is expanded in Equation 2.8. This form of representation transforms the evaluation of an MLP from one neuron at a time to one layer at a time and is typically the form used in most, if not all, AI inference operations [5].

$$W \times X + B = Y \tag{2.7}$$

$$\begin{bmatrix} w0 & w1 & w2 & w3 \\ w4 & w5 & w6 & w7 \\ w8 & w9 & w10 & w11 \end{bmatrix} \cdot \begin{bmatrix} x0 \\ x1 \\ x2 \\ x3 \end{bmatrix} + \begin{bmatrix} b0 \\ b1 \\ b2 \end{bmatrix} = \begin{bmatrix} y0 \\ y1 \\ y2 \end{bmatrix} \tag{2.8}$$

Other kinds of operations are used in AI depending on the use-case and type of output needed or form of input used. some of these operations are explored in more depth in the following section.

2.3 Artificial Intelligence Operations

A typical AI model for inference tasks could include many types and numbers of layers; however, the relationship between the layers is abstracted into sets of operations ranging from scalar multiplication and addition to matrix and vector multiplications that are carried out by the Processing Elements (PEs). Additionally, activation functions, quantization and de-quantization operations, and resizing operations can be performed. AI models can utilize any association between these operations, but Multi-layer perceptrons (MLPs), Convolutional Neural networks (CNNs), and Transformers are the most complex and highly parallel operations that most AI models utilize, where, based on general use cases, MLPs seem to be used for facts and relationships, CNNs for extracting information and patterns of image inputs, and transformers for providing context for input tokens for generative models.

2.3.1 Multi-Layer Perceptron (MLP)

A typical Processing Element (PE), or core in an AI accelerator, is designed to perform the computations of one or more neurons in a DNN or MLP layer, represented by a circle in Figure 2.4 with the destination $\mathbf{L_{m,n}}$ and connected by the gray lines that represent the pre-trained weights, performing matrix multiplication, accumulation, and an activation function. Depending on the chosen dataflow strategy, memory access can differ between implementations, and operation order and latency of execution will differ. All neurons in

[2] Uppercase notation implies a matrix of elements of the same letter with lowercase notation

a layer are computed in parallel before moving to the next layer, as the output of each layer is the input of the next layer.

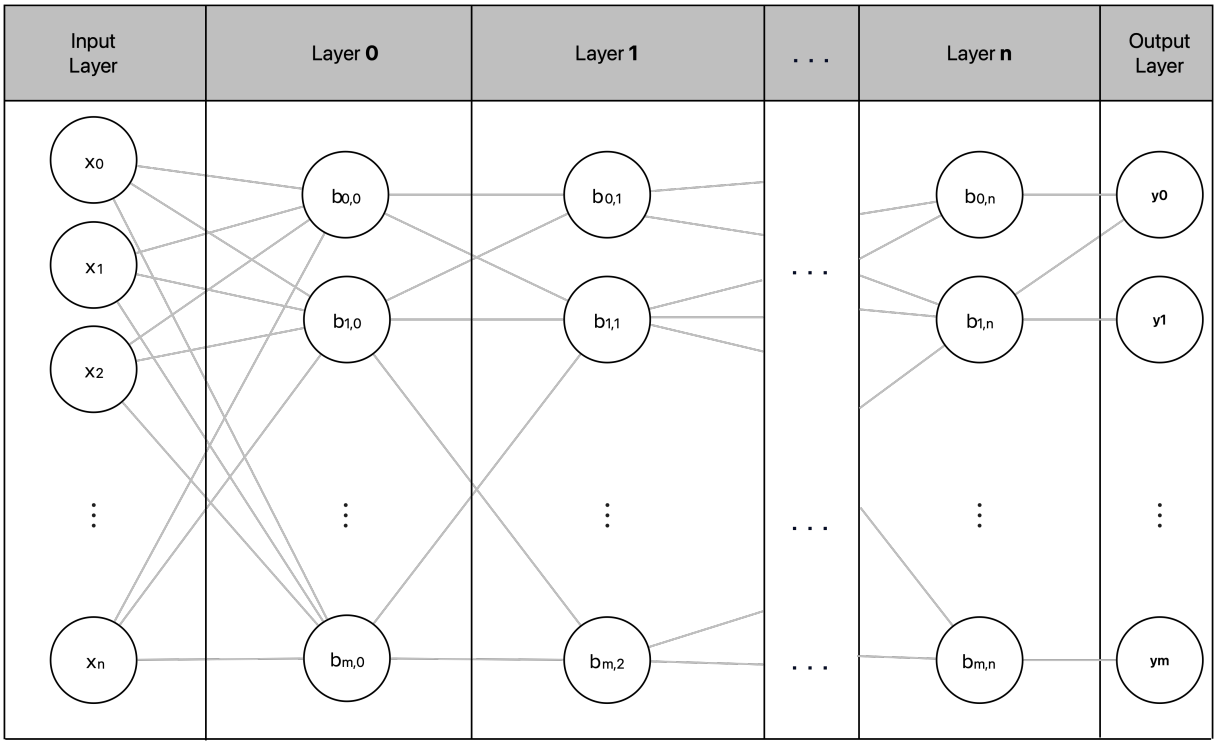


Figure 2.4: Network Layers in a DNN [5] where x_n represents the inputs, y_m represents the output, $b_{m,n}$ represents a neuron with certain bias b, and the lines between the neurons represents the weights of the model.

Matrix multiplication for the use of neural networks can be abstracted into the dot product between a matrix $W_{m\times n}$ that contains the trained weights of a model, and an input vector $X_{n\times 1}$ that results in an output vector $Y_{m\times 1}$. This is shown in Equation 2.9, where b_m represents the bias of each node in a layer. y

$$y_{m,1} = (\sum_{k=0}^{n-1} w_{m,k} \cdot x_{k,1}) + b_m \tag{2.9}$$

Equation 2.9 is applied to each layer of the network depicted in Figure 2.4. Instead of element-wise multiplication, it is typically evaluated as a whole layer performing matrix multiplication, as described in Equation 2.7 shown below, and moving to the next layer in the same manner shown in Equation 2.1. An activation function, such as the one shown in Equation 2.5 and Equation 2.6, is applied to each node in the layer.

$$Y = W \times X + B \tag{2.7}$$

Since most AI Accelerators have a limited number of PEs, a full layer may not be loaded or processed at once. Tiling or splitting into batches is required. This is carried out by splitting each layer into batches of sizes that match the available PEs. Batching will be discussed in Section 2.4.

2.3.2 Convolutional Neural Networks (CNN)

Convolution involves a 2D input feature map, $I_{w \times h}$, a 2D filter, $F_{f_w \times f_h}$, that traverses the input in a stride of size S. During this process, the filter performs dot product multiplication with an equally sized subsection of the input, resulting in a 2D output feature map, $O_{o_w \times o_h}$. This operation is shown in Equation 2.10.

$$O_{o_w,o_h} = \sum_{m=0}^{f_w-1} \sum_{n=0}^{f_h-1} I_{(o_w+m),(o_h+n)} \cdot F_{m,n} \tag{2.10}$$

$$1 \leq o_w \leq \frac{w - f_w}{s} + 1 \tag{2.11}$$

$$1 \leq o_h \leq \frac{h - f_h}{s} + 1 \tag{2.12}$$

The convolution operation shown in Equation 2.10 works by applying element-wise multiplication of a filter and a section of the input feature map, and adding the values of this set of multiplications. The whole process is then reapplied in a sliding-window manner that moves S rows or columns each step, producing an output feature map of the sizes based on Equation 2.11 and Equation 2.12. Convolutional neural networks (CNNs) use a set of filters, sometimes at every layer of the NN, to extract different features from the input data. For example, edge detection filters 2.13 highlight the differences in pixel values between the neighboring pixels, blob detection filters detect circular shapes, and line detection filters for vertical, horizontal, and diagonal lines. The used filters depend on the use case of the CNN and which tasks they perform.

$$F_{edge} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{2.13}$$

Similarly to matrix multiplication operations, a full CNN may, and usually is, not loaded fully into an accelerator, and needs to be split into batches matching the amount of PEs available. In order to match the resource availability of the hardware, batching of operations is used and is discussed in Section 2.4.

2.3.3 Transformers

Finally, Transformers, first described in [18], employ a set of different matrix multiplications to process input tokens, such as a textual sentence, to generate a specific output, like a textual paragraph.

Each input token for a transformer is represented by a high-dimensional vector that encapsulates its position in a higher-dimensional space. Each location in this higher-dimensional space embodies a unique characteristic of the input token. Transformer operations can be conceptualized as four distinct steps.

- **Embedding**: The higher-dimensional vectors representing the tokens are multiplied each by an embedding matrix W_E to produce the embedded vectors $\vec{E}$ shown in Equation 2.14 where n represents the context size of the network, i.e., how many tokens can be processed for one inference. The embedding matrix W_E is a pre-trained set of weights that have semantic information related to the set of tokens that it can process.

$$\vec{E}_n = \vec{Token}_n \cdot \mathrm{W}_E,\ 0 \leq n \leq \text{context size} \tag{2.14}$$

- **Scaled Dot-Product Attention**: Works by creating an attention pattern that relates the tokens to each other in a semantic way, rather than treating the tokens as independent variables, for instance relating adjectives to nouns in a textual token input. This is done by calculating the query vectors $\vec{Q}$ and key vectors $\vec{K}$ for each of the embedded vector $\vec{E}$ shown in Equation 2.15 and Equation 2.16 by multiplying the embedded vector with a query matrix and a key matrix that are pre-trained before inference.

$$\vec{Q}_n = \vec{E}_n \cdot \mathrm{W}_Q,\ 0 \leq n \leq \text{context size} \tag{2.15}$$

$$\vec{K}_n = \vec{E}_n \cdot \mathrm{W}_K,\ 0 \leq n \leq \text{context size} \tag{2.16}$$

The query vectors can be thought of as asking questions that are answered by their relevance to the key vectors. This is done by a dot-product between each query vector and all the key vectors as shown in Equation 2.17, then applying a soft-max function on the outputs of each query multiplication (the columns in Equation 2.17) with the keys. The scalar multiplication of the dimension of the key vector stabilizes the distribution of the results in case small values result in the same column as huge values that skew the distribution [18].

$$\begin{bmatrix} \vec{Q}_0 \cdot \vec{K}_0 & \vec{Q}_1 \cdot \vec{K}_0 & \cdots & \vec{Q}_n \cdot \vec{K}_0 \\ \vec{Q}_0 \cdot \vec{K}_1 & \vec{Q}_1 \cdot \vec{K}_1 & \cdots & \vec{Q}_n \cdot \vec{K}_1 \\ \vdots & \vdots & \cdots & \vdots \\ \vec{Q}_0 \cdot \vec{K}_n & \vec{Q}_n \cdot \vec{K}_n & \cdots & \vec{Q}_n \cdot \vec{K}_n \end{bmatrix} \cdot \frac{1}{\sqrt{d_k}} \tag{2.17}$$

The result of applying softmax on each column in Equation 2.17 can be abstracted into Equation 2.18 [18].

$$Attention(\vec{Q}, \vec{K}, V) = softmax(\frac{\vec{Q} \cdot \vec{K^T}}{\sqrt{d_k}}) \cdot V \tag{2.18}$$

The value vector $\vec{V}$ used in Equation 2.18 is the result of the multiplication of the embedded vector $\vec{E}$ with the pre-trained value matrix $\mathbf{W_v}$, shown in Equation 2.19 where the row values of $\vec{V}$ are multiplied to the row values of each column of the result of softmax to Equation 2.17.

$$\vec{V}_n = \vec{E}_n \cdot \mathrm{W}_V,\ 0 \leq n \leq \text{context size} \tag{2.19}$$

Now, all the column values in each column are added to produce the higher-dimensional change needed for each original embedded vector represented by $\Delta\vec{E}$ to give contextual information to the embedded vector. This is shown in

$$\vec{E'}_n = \vec{E}_n + \Delta\vec{E}_n,\ 0 \leq n \leq \text{context size} \tag{2.20}$$

- **MLP**: The same operations described in Section 2.3.1 are used to provide factual information for the outputs of the Scaled Dot-Product Attention.
- **Unembedding**: Similar to embedding, $\vec{E'}$ is stripped from the higher-dimensional contextual information to provide an output token in which the position of the token in the output is contextually understandable by humans. This is done by the multiplication of $\vec{E'}$ with an unembedding matrix W_U shown in Equation 2.21

$$\vec{Token}_n = \vec{E'}_n \cdot \mathrm{W}_U,\ 0 \leq n \leq \text{context size} \tag{2.21}$$

This describes only one head of attention; however, multiple heads of attention can be applied by repeating Scaled Dot-Product Attention and MLP operations multiple times, where each head is responsible for a certain contextual understanding of the input tokens.

2.4 Batching of Operations

Batching and stitching of operations are used to overcome the processing or memory limitation of hardware accelerators, In batching, a matrix multiplication or convolution operations are split into smaller sections that can be handled by the accelerator. this split is directly related to the weights used in a model and whether they fit into the memory or the processing unit of the accelerator.

2.4.1 Batching in Matrix Multiplication

Splitting matrix multiplication can be done in multiple ways, such as:

1. Row or Column Blocks.
2. Strassen's Algorithm for square matrices.
3. Distributed or Parallel Multiplication.
4. Recursive Splitting.

As an example, Row or Column Blocks method is when the weights or input matrices are split into smaller batches, and each batch is processed individually either serially by the same processing unit or in parallel by multiple processing units. The results of each batch are stitched together to produce the final result.

For instance, a matrix A of size $m \times k$ and a matrix B of size $k \times n$ can be split in the following way:

$$A_{m\times k} = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}, B_{k\times n} = \begin{bmatrix} B_1 & B_2 \end{bmatrix} \text{ where}$$

- A is split horizontally into A_1 and A_2 each of size $\frac{m}{2} \times k$.
- B is split vertically into B_1 and B_2 each of size $k \times \frac{n}{2}$.

The result matrix $C_{m,n}$ is computed by performing multiple matrix multiplications based on the spatial location of the new sub-matrices.

$$C_{m,n} = \begin{bmatrix} A1 \times B1 & A1 \times B2 \\ A2 \times B1 & A2 \times B2 \end{bmatrix}$$

2.4.2 Batching in Convolution

Convolution and pooling operations are typically performed by sliding a filter over the input data and performing a dot product between the filter and the input data. Pooling is very similar to convolution by performing a specific operation on the kernel, such as taking the maximum value of a set of values, and sliding over in the same way as convolution.

Convolution on an input, typically an image, known as a feature map, with one or multiple filters produces one output feature map. A common notation for a convolution operation is to represent the number of batches, the number of channels, the height, and the width of the filter in a tuple $\langle B, C, f_w, f_h \rangle$ which means the application of B number of convolution operations, each operation has C channels (matrices) on the input feature map of the same number of channels, where each channel is only convoluted with the corresponding channel in the filter, and the resulting C matrices are summed up to produce the output feature map. To unpack this further, taking an example of convolution of an input feature map of 3 channels, such as the Red, Green, and Blue (RGB) channels of an image, with one filter with three channels $\langle B = 1, C = 3, f_w, f_h \rangle$; Each channel from the input feature map will be convoluted with the corresponding channel in the filter, and the resulting three matrices will be summed up to produce the output feature map. This is shown in Figure 2.5 where the input feature map is of size $m \times n$ and the filter is of size $f_W \times f_H$. The output feature map is of size $m - f_w + 1 \times n - f_h + 1$.

Convolution of one batch (filter) produces one channel in the output feature map regardless of how many channels are in the input feature map. However, current software implementations of convolution operations in AI codes allow multiple batches of convolution on the same input feature map to extract multiple features in one operation; each batch will produce one channel in the output feature map. This is shown in Figure 2.6 where B batches of size $f_w \times f_h$ are applied to the input feature map of size $m \times n$, and the resulting output feature map is of size $m - f_w + 1 \times n - f_h + 1$. The input feature map and the filters have 3 channels each following the typical RGB image format but are not limited to 3 channels.

To summarize, A convolution operation can be described in a tuple $\langle B, C, f_w, f_h \rangle$ where:

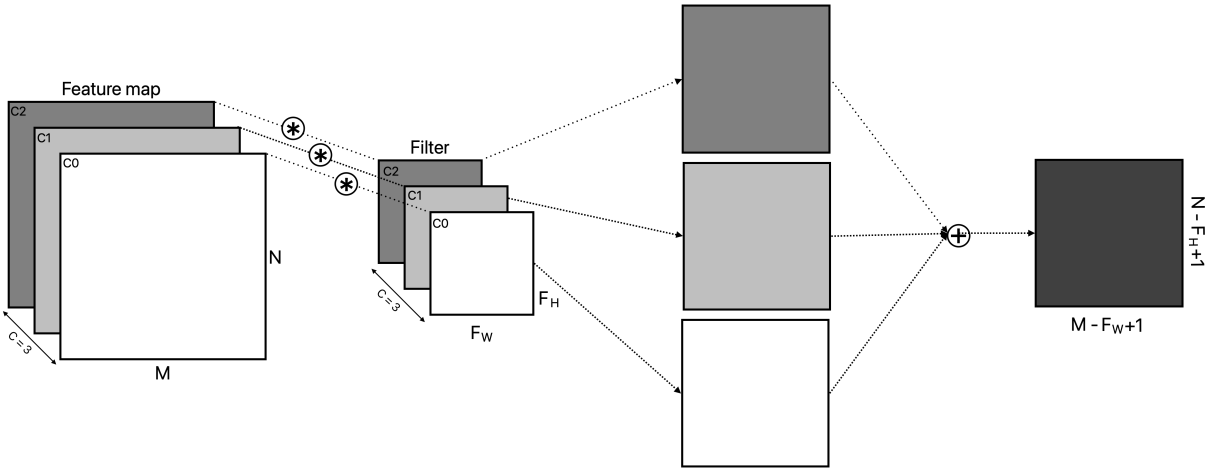


Figure 2.5: Multi-channel convolution of one batch: Each channel requires a filter, producing one output feature map with one channel.

- B is the number of batches (or filters) applied to the input feature map.
- C is the number of channels in the input feature map and the filter (must be equal).
- f_w is the width of the filter.
- f_h is the height of the filter.

The number of channels in the input feature map must be equal to the number of channels in the filter. The steps to perform a convolution operation are as follows:

1. Each batch represents one filter.
2. Each channel C in the input feature map is convoluted (using matrix multiplication) with the corresponding channel C in the filter.
3. each batch will produce C number of channels that will be added to each other to produce one output feature map.
4. every output feature map produced by each batch will represent one channel in the output feature map.

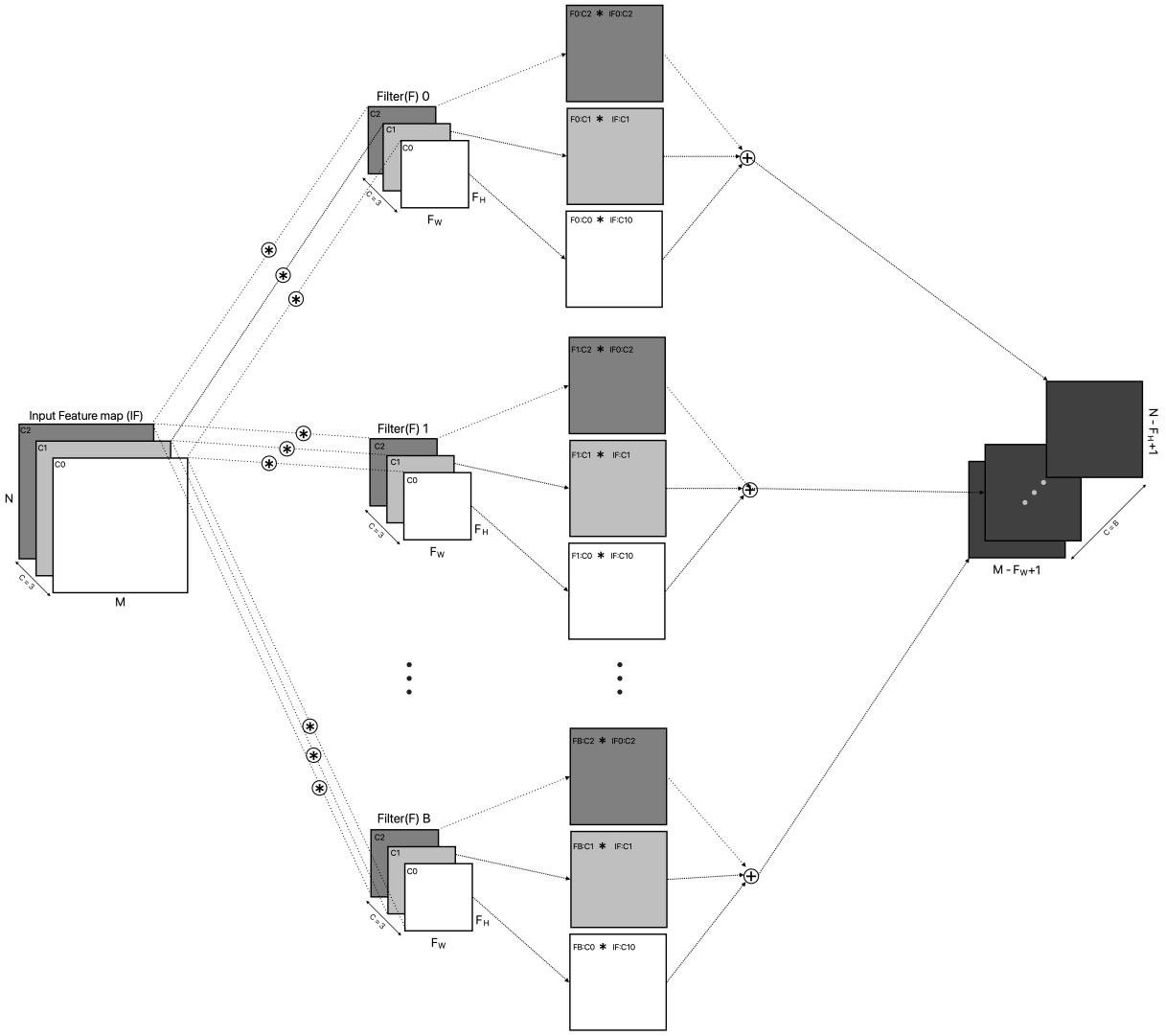


Figure 2.6: Convolution of B batches (or filters) of an input feature map of 3 channels, each batch has 3 filters, and produces an output feature map of B number of channels.

2.4.3 Convolution as Matrix Multiplication

Expressing convolution as matrix multiplication is crucial for accelerators that can only perform matrix multiplication operations and described in this subsection.

For a two-dimensional input feature map $I_{w \times h}$ and a two-dimensional filter $F_{f_w \times f_h}$, The output feature map $O_{o_w \times o_h}$ would be of a size that follows Equation 2.11 and Equation 2.12 and can be converted into matrix multiplication, specifically Toeplitz matrix multiplication [5]. The steps to convert a convolution operation into matrix multiplication are as follows:

1. Convert every filter patch of size $f_w \times f_h$ from input feature map in a row-major fashion into a column in a new matrix Col_{col_w, col_h}, called column matrix sine each column represents a convolution window of the original feature map.

2. Convert the filter $F_{f_w \times f_h}$ is then into a row vector of size $f_w \cdot f_h$.

3. Perform matrix multiplication normally.

4. Reshape the output matrix into the output feature map $O_{o_w \times o_h}$.

For example, Assume an input feature map $I_{3\times3}$ and a filter $F_{2\times2}$ that moves in a stride S=1:

$$I_{3\times3} = \begin{bmatrix} a1 & a2 & a3 \\ a4 & a5 & a6 \\ a7 & a8 & a9 \end{bmatrix} , F_{2\times2} = \begin{bmatrix} f1 & f2 \\ f3 & f4 \end{bmatrix}$$

The resulting column matrix would be: $Col_{4\times4} = \begin{bmatrix} a1 & a2 & a4 & a5 \\ a2 & a3 & a5 & a6 \\ a4 & a5 & a7 & a8 \\ a5 & a6 & a8 & a9 \end{bmatrix}$

and the row vector would be: $F_{1\times4} = \begin{bmatrix} f1 & f2 & f3 & f4 \end{bmatrix}$

performing the matrix multiplication would result in a 1x4 matrix

$$\begin{bmatrix} f1 & f2 & f3 & f4 \end{bmatrix} \times \begin{bmatrix} a1 & a2 & a4 & a5 \\ a2 & a3 & a5 & a6 \\ a4 & a5 & a7 & a8 \\ a5 & a6 & a8 & a9 \end{bmatrix} = \begin{bmatrix} o1 & o2 & o3 & o4 \end{bmatrix}$$

which would be reshaped into a 2x2 output feature map.

$\begin{bmatrix} o1 & o2 \\ o3 & o4 \end{bmatrix}$ where, $o1 = f1 \cdot a1 + f2 \cdot a2 + f3 \cdot a4 + f4 \cdot a5$ and so on for the other elements.

The contents of the Fundamentals Chapter provide some overview to attempt to find answers to the first two research questions: whether WCET can be applied to AI Accelerators and what operations exist in AI inference. In order to estimate the WCET of an AI model, an appropriate execution model of the accelerator is needed, and with the type and description of operations used in the model, the flow facts can be inferred to set up appropriate analysis tools and methods for said model executing on an accelerator.

The next chapter, Related Work, goes over contemporary research that covers WCET in the context of AI accelerators, highlighting the different WCET approaches used and justifying which approach is going to be taken in this work.

3 Related Work

Research related to Worst Case Execution Time (WCET) and timing analysis related to modern multicore computing, Graphics Processing Units (GPUs), and AI accelerators is not as simple as instruction counting and high-level code analysis due to the designed nature of the highly parallel design and focusing on improving the average case performance of these systems. For instance, the main objective of a GPU was to accelerate graphics, in which every additional rendered frame is a bonus, and this artifact has evolved over to general-purpose GPU (GPGPU) usage.

Trends towards multicore computing stem out of the need to increase the performance of the system, in which increasing the clock frequency was the preferred method but seemed unrewarding to step over the 5GHz barrier for a single-core system, and thus the multicore route was taken. Similar to GPGPUs, very specific hardware accelerators for matrix multiplication that utilize a similar parallel nature of the GPU have been developed. Research relating to WCET of single or multicore CPU systems is established with methods and industries revolving around providing novel solutions to the problems. However, the same cannot be said about timing analysis and WCET of GPUs and AI accelerators, as the use of these devices is still, relatively, in its infancy. The following sections will discuss the current state of the art in WCET analysis for single and multicore systems, GPUs, and AI accelerators.

In this chapter, some papers related to the WCET and predictability of using AI accelerators are discussed. The approaches tend to split between GPU based accelerators and Systolic-based accelerators which will be discussed further in Chapter 4.

The authors in [11] describe the pitfalls of static analysis for GPU WCET estimation, in which static analysis requires an accurate hardware and execution model. Additionally, the authors identify incompatibility between the use of control flow graphs (CFG) with GPU analysis due to the branch divergence behavior of the GPUs, in which threads with different execution paths are not executed in parallel. The authors use a dynamic analysis approach to estimate the WCET of GPU kernels with instrumentation points and Instrumentation Point Graph (IPG) to capture the execution paths of the kernel from which the authors then create a CFG that allows them to estimate the WCET of the kernel. The authors used a cycle-accurate simulator to validate their results.

The authors of [6] agree with [11] regarding the incompatibility of static analysis with GPUs, they argue that the lack of information regarding scheduling and the nature of the microarchitectures of GPUs is the main reason for the incompatibility. The work in [6] introduces a WCET analysis tool based on Symbolic Execution (SE) and Genetic Algorithm (GA). The authors target Compute Unified Device Architecture (CUDA) kernels (which run only on Nvidia GPUs) and, similarly to [11], the authors focus on the branch divergence phenomenon. The authors aim to run a CUDA code with systematic synthetic

inputs that aim to cover all execution paths of the kernel as opposite to randomly generated inputs.

In [11, 6], branch divergence is taken into account for WCET. Branch divergence is a result of using one program counter register for multiple threads, causing threads that follow different branches in code to execute in a serial fashion, heavily affecting concurrency, timing analysis, and code prediction. However, modern Nvidia GPUs (any using the Volta or newer architecture) implement independent thread scheduling in which threads with different execution paths are executed in parallel regardless of which path each individual thread is following. This is implemented through an independent program counter for each thread as an upgrade from a shared single program counter [19].

Unlike the approaches in [11, 6], the authors of [8, 20] pay special attention to the shared data caches in estimating WCET of GPU processes.

In [8], the authors address the design choices of GPUs to improve average case execution, which heavily impacts WCET and predictability. They also address specific features that also affect predictability, such as out-of-order execution and dynamic scheduling; however, there is no concrete proof that Nvidia GPUs employ such techniques. The focus of the work in [8] is to improve predictability of the L1 data cache, which they claim to be a major issue for time predictability. They propose a compiler and a framework that reorders the load-store operations alongside the wrap scheduling. The authors claim that this would produce better performance too.

A purely static-analysis method of CUDA kernels is provided in [21] where a model that estimates delays based on Nvidia's Parallel Thread Execution (PTX) virtual instruction set is developed and then used alongside a simulation model. The authors of [21] take into account memory access patterns and virtual instruction delays.

The work in [20] focuses on systems that use integrated CPU and GPU dies and claims that the integrated systems have the potential to improve the performance of real-time applications. The authors focus on analyzing the access times and WCET of shared data cache between the integrated CPU and GPU, known as the Last Level Cache (LLC), which they claim can be used to estimate the WCET of the GPU. The authors in [20] conclude that the run-time behavior of the LLC is hard or impossible to predict statically due to run-time artifacts produced by the multicore system but claim that the results can be used to reduce the overestimation of WCET.

The paper [22] discusses the application of systolic arrays to enhance matrix multiplication efficiency in deep neural networks (DNNs). With the rise of AI models, such as OpenAI's Generative Pre-trained Transformer (GPT) with trillions of parameters, efficient matrix operations have become paramount. This study focuses on three primary systolic array data flows: Weight Stationary (WS), Input Stationary (IS), and Output Stationary (OS), and analyzes their energy consumption across various matrix configurations using a systolic array simulator. The findings reveal that selecting an optimal data flow based on matrix size can significantly reduce energy usage, paving the way for the design of energy-efficient DNN accelerators.

The paper in [22] highlights the limitations of traditional Von Neumann architectures for the use in matrix multiplication, which often lead to bottlenecks. Systolic arrays are designed to address these limitations by enabling simultaneous data flow and parallel

processing. By organizing data in spatial dimensions, systolic arrays can enhance computation speed and reduce energy demands. The study evaluates the energy efficiency of each data flow, demonstrating that energy savings are maximized when the smallest matrix dimensions are mapped to the systolic array's spatial dimensions. This research contributes significantly to the literature on DNN hardware optimization by providing insights into how hardware acceleration for matrix multiplication can be tailored to different neural network applications.

Snopce and Aliu in [23] discusses the use of systolic arrays for matrix multiplication, comparing different fanout configurations and their impact on data availability. It highlights the trade-offs between fanout and data availability, with higher fanout providing faster processing but requiring more data storage. The article also presents graphical representations and data availability diagrams for each fanout configuration. This paper analyzes the latency of 2D systolic arrays for matrix multiplication, considering different connection schemes. The lower bound of latency is determined to be n, considering the fixed Input/Output bandwidth and the number of input elements. The paper also discusses the four possible connection schemes for systolic arrays. Systolic arrays with fan-outs of 1, 2, 3, and 4 have specific latency and bandwidth requirements. The fan-out 8 array achieves the lowest latency of $n/2$, while the fan-out 2 array has the highest bandwidth of n^2. The analysis carried out shows that it is impossible to achieve a latency of n for matrix multiplication using systolic arrays. The work in [23] goes over some concepts including determining the lower bound of latency achievable with different connection schemes in the array. The authors define Data entry time, data delay time, and data processing time and derive a formula based on the fanout of the array. The work in [23] was an inspiration for the work in this thesis.

The summary of the papers and the issues they identify regarding WCET of GPU-based systems is shown in Table 3.1

Issue / Approach	[**11**]	[**6**]	[**8**]	[**20**]	[21]
WCET Analysis Approach	Dynamic Analysis	Symbolic Execution (SE)	Compiler Optimization	Empirical Cache Analysis	Static Analysis
Tools Used	IPG, Cycle-Accurate Simulator	SE, Genetic Algorithm (GA)	Custom Compiler Framework	Shared Last Level Cache (LLC) Analysis	PTX-based
Branch Divergence	√	√	×	×	√
Cache Behavior	×	×	√	√	×
Static Analysis	√	√	×	×	√
Provides Execution Model	×	×	×	×	√ based on PTX

Table 3.1: Comparison of WCET analysis approaches and issues in GPU-based systems. ×: Identified and/or not Addressed. √: Identified and/or Addressed.

Current approaches in GPU WCET literature tend to focus on one or some execution paradigms, but they lack a holistic approach that integrates execution cycles with other relevant factors like cache behavior and branch divergence. For instance, approaches described in [11, 6] don't employ static analysis due to the absence of an execution model. The approach in [21] is a purely static approach that develops a simulation model equivalent to a working execution model but does not consider actual instruction execution relying on the better documented virtual intermediate instruction set (PTX). On the other hand, the authors of [8, 20] exclusively focus on analyzing cache behaviors. For Systolic based

systems, on the other hand, the literature focuses on expressing the execution cycle of an operation using an operation-count approach, mainly multiply and accumulate (MAC) operation either individually or combined, and is claimed to be sufficient for WCET purposes.

The next chapter will categorize the AI accelerators and give an overview on the scheduling and tasks assignment.

4 Analysis of Hardware Accelerators

The basic mathematical concept of the AI operations used in AI was explored in the previous chapter alongside the basic concept of WCET as a way to pave the answer to the first two research sub-questions. However, a middle ground that needs to be discussed first is the architectural differences and scheduling of the hardware, before digging deeper into the execution model which will be tackled in Chapter 5.

There exists a black-box atmosphere in the competitive market of AI accelerators in which companies race each other to develop the most attractive AI accelerator, which heavily impacts research in the field of hardware characterization and WCET. This chapter attempts to find the general information needed to understand the different hardware and characterize said accelerators in categories that enable some form of abstraction for the deeper execution analysis performed in Chapter 5 This chapter will focus on higher-level execution style and scheduling rather than going deeper into the execution models of the accelerators, which will be discussed in Chapter 5.

4.1 Characterization of Accelerators

AI Accelerators is an umbrella term that covers different types of hardware designed or used to accelerate AI-related operations. As explored in previous chapters, matrix operations represent a majority of operations where the AI accelerator must be optimized to perform matrix operations. According to [5], AI-Specific computing devices are constructed of parallel compute units and high-bandwidth storage units, often organized in a 2D assembly to match the nature of matrices. Multiple versions of AI Accelerators exist on the market today as seen in Figure 1.1, based on:

- Low- to high-power consumption,
- use in edge or server applications,
- optimized for training or inference use cases,
- based on Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), or Application-Specific Integrated Circuits (ASICs),
- are optimized for specific operations, such as DNNs, CNNs, etc.,
- support different numerical precision, such as 8-bit, 16-bit, 32-bit, etc.

However, this cannot serve as an appropriate characterization for timing analysis as it is too broad and does not provide a deep understanding of the differences and similarities between the accelerators which is required for architectural analysis. A more thorough understanding is through the characterization of the hardware architectures, leading

to a detailed understanding of the execution model, memory and cache behavior, and instruction delays.

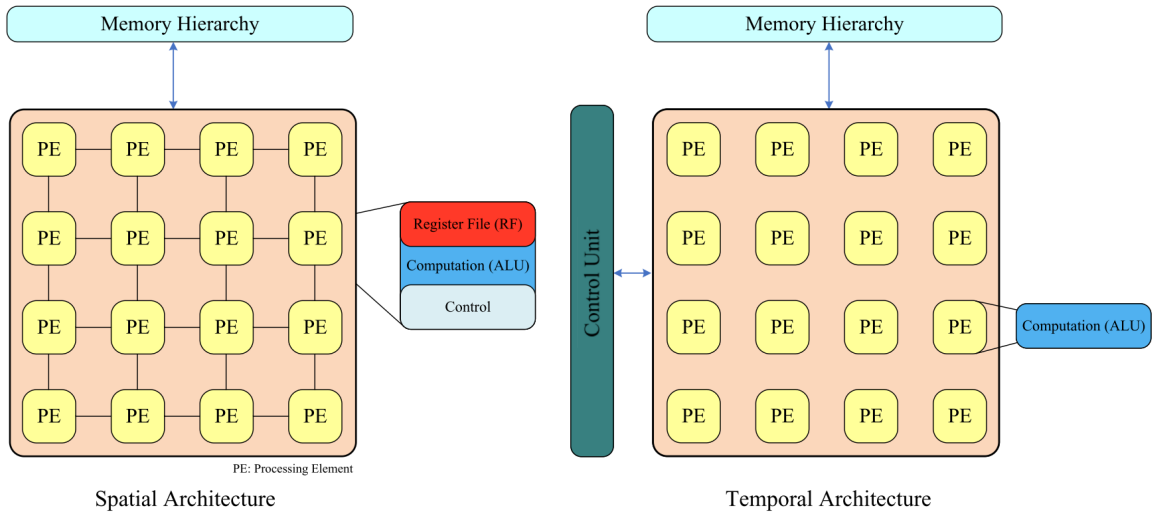


Figure 4.1: Assembly of Processing Elements (PEs) in Spatial (left) vs Temporal (right) Architectures [24].

Based on the research carried out in Chapter 3 and the work in [4], two major categories of AI Accelerators exist: **Temporal** and **Spatial** Architectures. Both of these architectures will be discussed in this chapter.

4.2 Spatial Architectures

Spatial architectures are specialized implementations of accelerators designed and implemented on Field Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs) from the ground up specifically optimized for matrix multiplication. These include Processing Elements (PE) that are interconnected to perform sub-operations of the main matrix multiplication operation and transmit data among each other, as shown in Figure 4.1 [24, 4]. Spatial DNN accelerators, shown in Figure 4.2, according to [4] typically consist of the following components:

- **Processing Elements (PEs)**:
  - Multiply and Accumulate Unit.
  - A small local register file: for each PE to store activations, weights, or partial summations.
  - Global buffers: to prefetch data from Memory.
- **On-chip buffers**
- **Controllers**
- **External Memory**

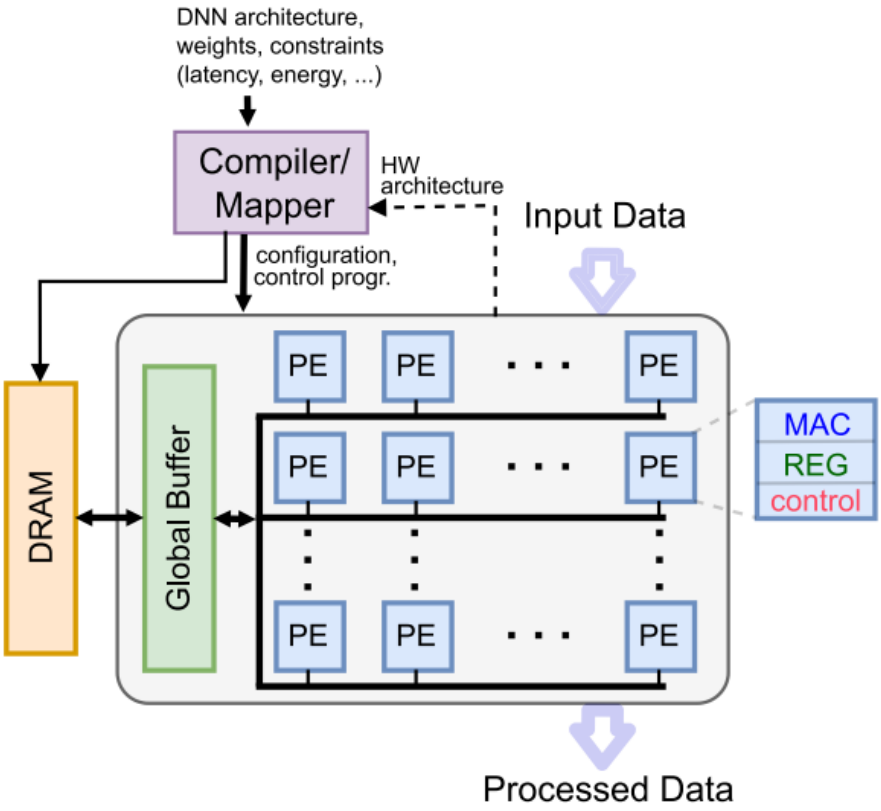


Figure 4.2: Typical assembly of Spatial DNN Architecture [4].

Each PE performs a set of operations, typically multiply, accumulate, and store operations, and transmits the data to the other PEs directly connected to it. The way this transmission of data is implemented is referred to as Dataflow, and there exist multiple different dataflow assemblies that are optimized for different use-cases.

4.2.1 Spatial Architectures Dataflow

Spatial, or systolic, accelerators use a dataflow strategy to minimize the impact of memory access, which is a significant bottleneck in temporal architectures when it comes to performing the highly parallel DNN operations. Memory access is one of the most time-consuming operations in DNNs on temporal architectures due to the high latency and power consumption of memory access and the hierarchical nature of the memories which is bypassed in spatial architectures. According to [4], dataflow refers to the computational order in which data is processed, stored, and reused. There exist four types of dataflow strategies [4, 24], namely:

- **Weight Stationary (WS)**: In this strategy, the weights are fixed and stored in the PE, typically in the register files in each PE. All operations that require the loaded weight are performed on the PE. This, in turn, reduces the number of memory accesses for the weights, but partial sums will have to be stored in a local memory where size could be limited, and thus the number of operations may need to be limited. [5, 25]

- **Output Stationary (OS)**: In this strategy, the output, or partial sums, is stored in the PE and accumulated as the data flows through the PE.
- **No Stationary (NS)**: Also known as no local reuse, where nothing is stored on the chip, and all data is fetched from the DRAM on demand.
- **Row Stationary (RS)**: WS and OS alone fail to take into account all optimizations at once. RS can be thought of as a combination of WS and OS, where weights are still stored in the PE, but the partial sum is calculated through the rows of the matrix. This is done by converting the matrices into one-dimensional arrays and performing the operations on the rows. This allows for the weights to be reused across the rows, and thus reduces the number of memory accesses to the DRAM.

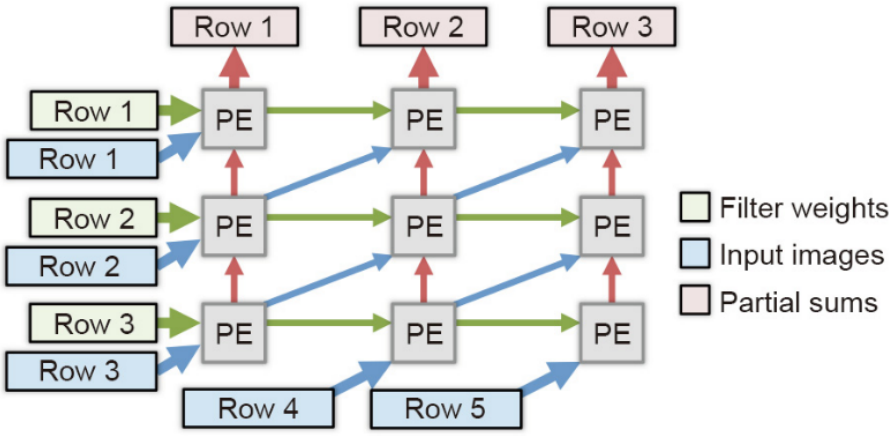


Figure 4.3: Typical assembly of Row Stationary Dataflow [5].

Typically, a DNN requires a large memory footprint to store the weights, activations, and partial sums, and [5] claims that Row Stationary, shown in Figure 4.3, is the most efficient dataflow strategy for DNNs. However, the choice of dataflow strategy is dependent on the DNN model and the hardware architecture. No Stationary is used when die area needs to be reduced, and this is usually done by eliminating the local register files [24]. However, DNN models are typically large and cannot be loaded fully into the AI Accelerators, and thus tiling is required [4], where chunks of data that can fit onto the AI Accelerator are loaded and processed at a time. This can be optimized by using the dataflow strategies mentioned above to facilitate memory reuse and reduce access times to DRAM.

4.3 Temporal Architectures

Temporal Architectures describe a set of architectures designed for general-purpose computing, but can perform AI Acceleration tasks with the right algorithms and software. Typically, CPU- and GPU-enabled systems fall under this category [24, 4]. CPUs are general-purpose processors that can perform a wide range of tasks, but are not generally optimized for AI operations. GPUs, on the other hand, are designed to perform parallel operations and are optimized for executing multiples of the same operation in parallel, which suits the nature of AI operations. Temporal Architectures typically employ a set of

Arithmetic Logic Units (ALUs) with fixed connections and memory hierarchies that are optimized for general-purpose computing [24, 4]. The ALUs are capable of performing a wide range of operations. Temporal architectures provide parallelism through the use of Single Instruction Multiple Data (SIMD) commonly found in CPUs and Single instructions multiple threads (SIMT) commonly found in GPUs [24]. Due to the memory hierarchies, ALUs in temporal architectures cannot communicate directly with each other, but by moving data across the different memory hierarchies [24, 4], see Figure 4.1, which can result in a bottleneck in the data flow. According to [24], CPU cores are the least used of the temporal architectures when it comes to inference or training due to the low number of processing cores and thus the number of processes that can occur in parallel. GPUs, on the other hand, are more commonly used for AI operations due to the high number of cores and the ability to perform parallel operations. Thus, as an example of a Temporal Architecture, focus will be diverted away from CPU implementations and more towards GPU implementations, specifically Nvidia's GPUs, since they hold 92% of the market share of GPU sales for AI-related use [26].

4.3.1 Nvidia GPUs: Temporal Architecture Example

Originally driven by the demand for real-time 3D graphics accelerators, Nvidia's Graphics Processing Units (GPUs) have evolved into a highly parallel, multithreaded, and many-core processor with immense computational power and high memory bandwidth. [19] GPUs shine in tasks where the same operation is performed on many data points in parallel, such as matrix multiplication in DNNs. Used as a coprocessor, a GPU performs a single instruction on multiple threads (SIMT). This is done by programming the operation as a function called a kernel, which then gets assigned to blocks and grids as an organization method. Kernels are executed as warps, which are the lowest level of schedulable objects and represent a set of threads (one instance of a kernel) that execute concurrently on one streaming multiprocessor (SM). [19] Each SM contains a set of Arithmetic Logic Units (ALUs) and other specialized cores that perform the operations on the threads in the warp. In an abstract overview, all threads in each warp execute the same instruction but on different data. This is done to minimize the number of cycles required to execute the instruction and thus increase the throughput of the GPU. A typical assembly of an SM in an Nvidia GPU is shown in Figure 4.4.

The assembly of a relatively modern Nvidia GPU, such as the Ampere or Hopper, consists of multiple **GPU Processing Clusters (GPC)** each consisting of multiple **Texture Processing Clusters (TPCs)** where every TPC have two **Streaming Multiprocessor (SM)**. The types of cores used in each SM is what usually differs the architectures from each other. For instance, the Ampere SM [27], showed in Figure 4.5 consists of

- 64 cores that handle 32bit Floating point operations,
- 64 cores that handle 32bit integer operations,
- 32 cores that handle 64bit floating point operations, and
- 4 cores that handle tensor (matrix) operations.

The capabilities and use of the Nvidia instructions is described through a virtual instruction set architecture (ISA) called Parallel Thread Execution (PTX) that enables the use of

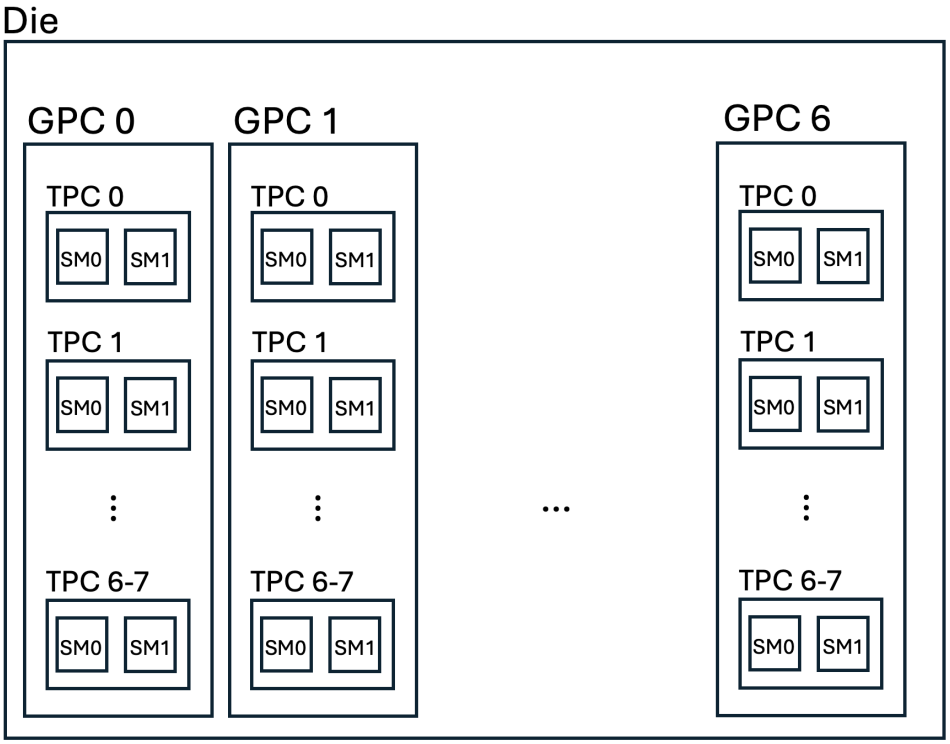


Figure 4.4: Structural representation of compute assembly of Nvidia Ampere Die [27].

Nvidia GPUs as general purpose coprocessor [19]. On top of PTX, a C++ language extension called the Compute Unified Device Architecture (CUDA) is developed, which facilitates a higher level of programming abstraction. Code written in CUDA gets translated into the PTX ISA and then into the actual machine code, known as Streaming Assembler (SASS) instructions that run directly on the GPU. It is claimed that PTX is used for better optimization and compatibility across different GPUs [19]. Nvidia PTX ISA describes a set of instructions that can be executed on the GPU and can be clustered into the following categories:

- Integer Operations,
- Extended-Precision Integer,
- Floating-Point,
- Half Precision Floating-Point,
- Comparison and Selection,
- Half Precision Comparison,
- Logic and Shift,
- Data Movement and Conversion,
- Control Flow,
- Parallel Synchronization and Communication, and
- Warp Level Matrix Multiply-Accumulate.

Figure 4.5: Nvidia Ampere Streaming Multiprocessor [27].

The Programming and execution model and of an Nvidia GPU will be discussed in Chapter 5, but the important distinctions of Kernel, Thread, Block, and Warp on an abstract level are identified in Table 4.1 since they also impact scheduling.

This subsection considers the scheduling of higher-level kernels and blocks rather than at the warp level, which will be discussed in Chapter 5. Examples of experiments and analysis done by [28] will be used to further explain the scheduling of Nvidia GPUs and the dataflow of operations.

4.3.1.1 Scheduling of Nvidia GPUs

To understand scheduling on the Nvidia GPU, which will be referred to as device, a distinction between two types of CUDA operations is needed; **Thread Operations** and **Memory Operations**, where thread operations are represented in a **kernel** that is scheduled to run in a highly parallel fashion. Kernels are assigned by the programmer

Term	Definition
Kernel	A function produced by the programmer in CUDA or PTX that is intended to run on an Nvidia GPU. In this chapter, a Kernel refers to the kernel itself in addition to the invocation parameters (how many blocks and how many threads per block.)
Thread	An instance of a compiled kernel that is currently loaded to the GPU. It is the smallest unit of execution where each thread contains the same code as the other
Block	defines a group of threads that run on one SM, blocks can be represented in one-, two-, or three-dimensional representation.
Warp	A fixed group of 32 threads that get issued the same instruction at the same time in one SM. It is the smallest schedulable object.

Table 4.1: Terms definitions used in scheduling in CUDA.

into the number of threads per block, the number of blocks per grid, and how many grids are needed. Threads, Blocks, and Grids define the structure of the parallel execution of the kernel but not the scheduling. Memory Operations, on the other hand, happen before and after launching the kernel to move data from the system memory, which will be referred to as host, to the device memory. All of these operations get associated with a **stream** which is a sequence of operations that are executed in order. The streams are programmer-defined, but if none is defined, all operations automatically get associated with the default **Null stream** [28].

All operations from all streams get split into two different queues, the **Compute Queue** and the **Memory Queue**. The Compute Queue is responsible for scheduling the kernels, and the Memory Queue is responsible for scheduling memory operations. [28]. The order in which the operations get assigned to the stream depends on the invoking time of the host code. However, the order in which the operations get to the queues depends on the host invoke time, source stream, and priority of the stream. For dispatching from the queues, a general understanding of schedulable entities follows the following:

- For every SM on the GPU, typically a maximum of 2048 threads can be dispatched at a time, with a maximum of 1024 threads per block.
- A block is dispatched only to one SM and cannot be split between multiple SMs.
- blocks are not necessarily dispatched concurrently, and depend on resource availability, which is different across the GPUs:
  - Shared memory availability per block,
  - Register availability per thread, block, or SM.

in [28], scheduling rules are summarized for all streams except the Null stream. And are as follows for enqueuing to **Compute Queue**:

- A kernel is enqueued to the associated non-null stream when the host code invokes the kernel.

- A kernel is enqueued onto the compute queue when it reaches the head of the not-null stream.
- Since null stream is not used, only blocks of the kernel at the head of the compute queue are eligible to be assigned without pre-emption.
- one or some blocks of the kernel at the head of the compute queue is eligible to be assigned only if its resource constraints are met (threads available, shared memory, and registers).
- a kernel is dequeued from the compute queue once all blocks are assigned.
- a kernel is dequeued from the stream once all blocks have finished executing.

And for enqueuing to **Memory Queue**:

- A memory copy operation (of either direction) is enqueued to the associated non-null stream when the host code invokes the kernel.
- A memory copy operation at the head of the stream is eligible to be assigned to the memory queue.
- A memory copy operation is dequeued from the memory queue once it is finished.
- A memory copy operation is dequeued from the stream once the copy operation has finished.

4.3.1.2 Scheduling without priorities nor Null Stream

An example of scheduling from [28] with kernels to be assigned to an Nvidia GPU with one copy engine and two streaming multiprocessors, where the kernels invoked by two different host codes assigning kernels to three streams. The kernels are described in Table 4.2.

Kernel	Launch info	Start Time (s)	Number of Blocks	Threads per block	Shared memory per block
K1	CPU 0, Stream S1	0.0	6	768	0
K2	CPU 0, Stream S1	0.0	2	512	0
K3	CPU 0, Stream S1	0.0	4	256	0
K4	CPU 1, Stream S2	0.2	4	256	32KB
K5	CPU 1, Stream S3	0.4	2	256	32KB
K6	CPU 1, Stream S2	2.8	2	512	0

Table 4.2: Scheduling example of kernels on Nvidia GPU [28].

In Figure 4.6 (a), CPU 0 invokes kernels K1, K2, and K3 to stream S1, and CPU 1 invokes kernels K4, K5, and K6 to stream S2 and S3. The kernels are assigned to the compute queue and memory queue based on the rules described above. K1 (in grey) is dispatched to the GPU first, 4 kernels are executed but since each kernel has 768 threads, only 2 blocks are executed at a time on each SM. The dispatching will have to wait since the other two

blocks cannot fit on the SM due to lack of thread availability (maximum 2048 threads per SM). In Figure 4.6 (b), the two remaining kernels from K1 are dispatched to the SMs, and K4 is dispatched next, and since K4 has 256 threads per block, 4 blocks are dispatched to the SMs. Even though there are still some threads available to execute K5, and it is at the head of the compute queue, K5 would not be dispatched due to lack of shared memory resource. In Figure 4.6 (c), K5 is dispatched to the SMs, and K6 is dispatched next, and since K6 has 512 threads per block, 2 blocks are dispatched to the SMs. Both copy operations in Stream 3 and 1 are still blocked until the blocks are finished executing. In Figure 4.6 (d), all blocks from K2 and K5 have finished, and the copy operations are at the head of their streams and can be assigned. In Figure 4.6 (e), all blocks from K3 and K6 have finished, and the copy operations are at the head of their streams and can be assigned [28]. The operations described in this example are visualized in Figure 4.7.

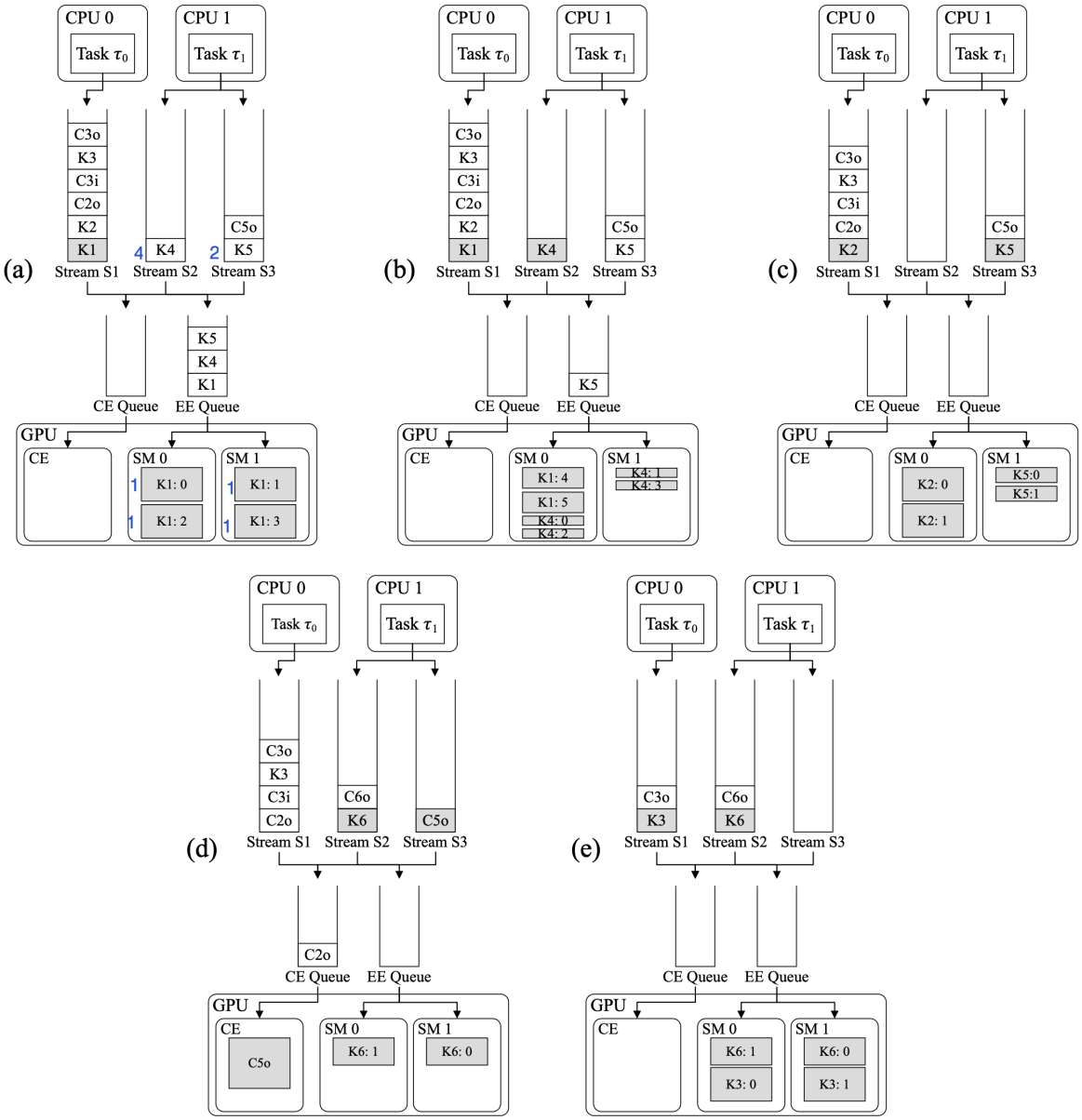


Figure 4.6: Scheduling of kernels in Table 4.2 on Nvidia GPU with two SMs [28].

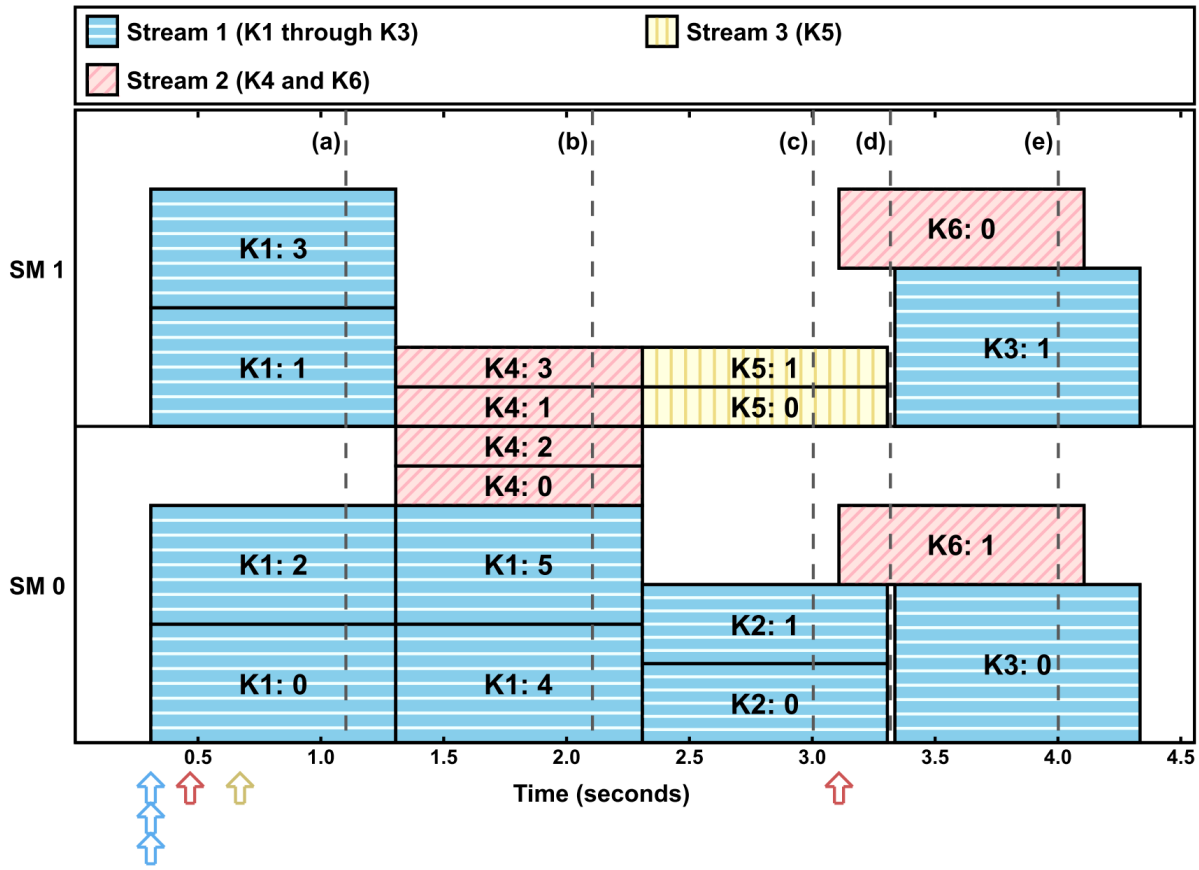


Figure 4.7: Execution timeline of kernels in Table 4.2 on Nvidia GPU [28] with two SMs.

4.3.1.3 Scheduling with Null Stream

The previous example excluded the null stream (S0), which can be thought of as the default stream when the programmer does not assign kernels or operations to a specific stream. According to [29], Null Stream should not be used if concurrency is required across multiple blocks of kernels with different user streams.

In [28] it is concluded that **Kernels at S0 only run if all other streams are empty or kernels at other streams are launched after that of S0 and form an execution barrier to any kernel invoked after it.**

An example of scheduling of Kernels using both the Null stream and user-defined streams are described in Table 4.3 and shown in Figure 4.8.

Kernels K2 and K5 were submitted to the null stream, but K2 did not get scheduled. Even though resources were available, since K2 was submitted after K3 and K4, and thus formed a barrier to the scheduling of K2. K5 was not scheduled until both K3 and K4 had finished since they were invoked before K5. [28]. The operations described in this example are visualized in Figure 4.8.

Kernel	Launch info	Start Time (s)	Duration (s)	Number of Blocks	Threads per block
K1	Stream S1	0.0	1	6	768
K2	Null Stream	0.2	1	1	1024
K3	Stream S2	0.2	1	4	256
K4	Stream S2	0.4	1	4	256
K5	Null Stream	0.6	1	1	1024
K6	Stream S3	0.8	1	2	256

Table 4.3: Scheduling example with of kernels using the null stream on Nvidia GPU [28].

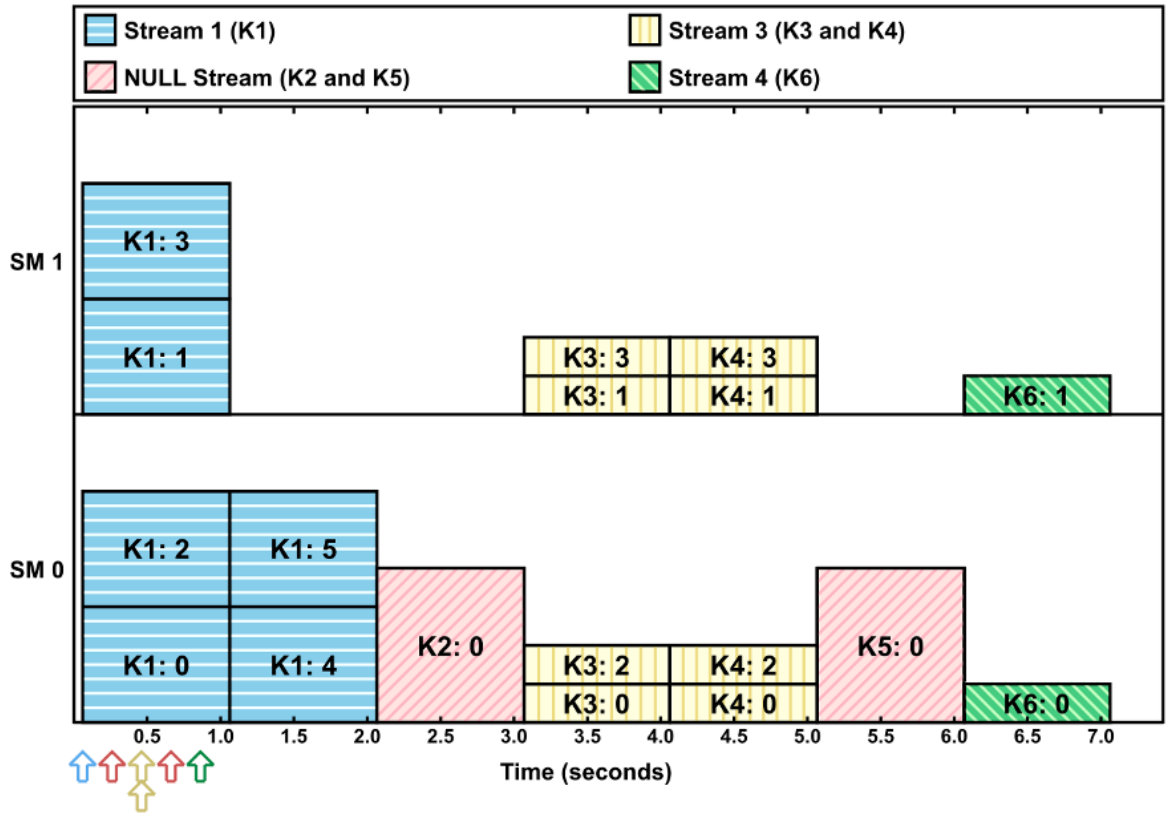


Figure 4.8: Execution timeline with Null Stream on Nvidia GPU [28].

4.3.1.4 Scheduling with Priorities

In [29], two priorities are described, namely **Low Priority** and **High Priority**, where not assigning a priority to a stream will default to low priority. The priority of a stream determines the order in which the operations are assigned to the compute and memory queues. Stream priorities are taken into account whenever a block finishes execution and a new block is to be scheduled. This also introduces preemption to scheduling. As shown in Figure 4.9, a low priority stream with 8 kernel blocks was scheduled ahead of time of two high priority streams. Half of the blocks were finished, and the other half were preempted by the high priority streams, and only after the high priority streams were finished, the low priority stream was scheduled again. [28].

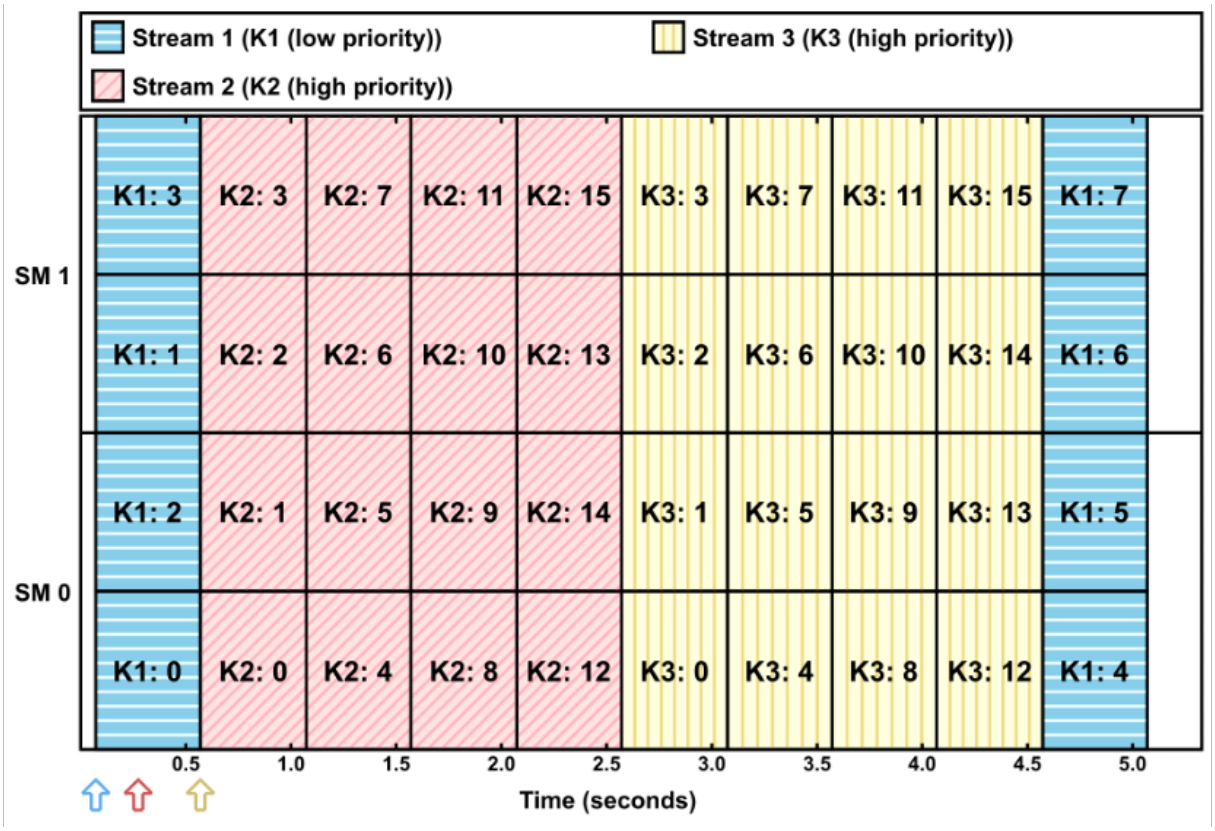


Figure 4.9: Execution timeline with Stream priorities on Nvidia GPU [28].

Scheduling rules are critical since a GPU can, a typically does, perform operations from multiple kernels invoked from different tasks concurrently when possible. A WCET method that analysis the individual kernels is required to accurately represent the execution time shown as the x-axis in scheduling graphs such as shown in Figure 4.9. Both scheduling and WCET depend on the physical capabilities of the GPU, such as shared memory, number of registers, etc., which will be further discussed alongside the WCET methods in the next chapters.

5 Design and Implementation

In this chapter, the execution model of AI accelerators is discussed in depth, based on the analysis carried out in previous chapters to create a suitable execution model for the worst-case execution time as an answer to the main research question **What is the worst-case execution time (WCET) of an inference operation executed on an AI accelerator?** This chapter covers Spatial architectures and a subsection of Temporal architecture, Nvidia GPUs, separately due to the significant differences between the two types of architectures.

5.1 Introduction

Pre-trained Artificial Intelligence (AI) models are trained and used as separate file formats that can be shared and loaded on different hardware, given compatibility. These files can be different based on the training framework used, such as Keras [30], PyTorch [31], Core Machine Learning (CoreML) [32], TensorFlow [33], among others. A model stores the sequence of operations of the model and the weights of each of the operations and then is loaded to be used on an input. Each hardware platform executes these operations in different ways for optimization purposes; for instance, Figure 5.1 shows a subset of operations from a pre-trained model that does not target any specific hardware platform, whereas Figure 5.2 shows the same subset of operations from the same model that targets the Apple Metal platform.

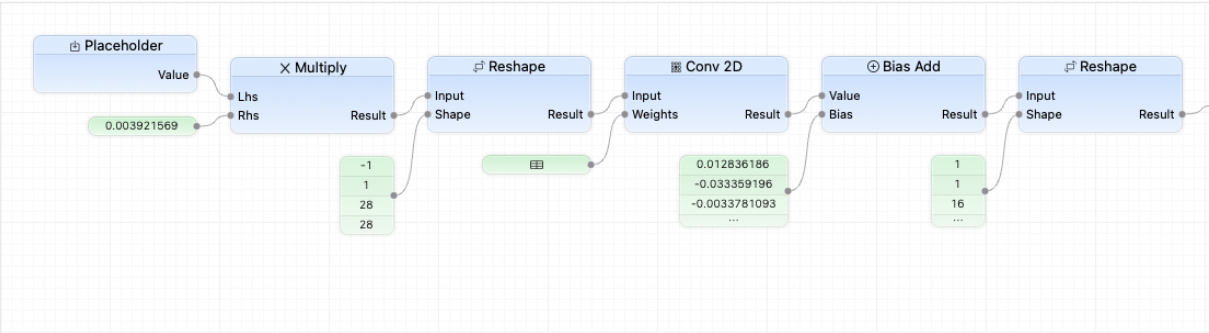


Figure 5.1: A section of an unoptimized pre-trained AI model.

Considering the variations in hardware-level implementation of one model, it is important to concentrate less on the model itself and more on the low-level implementation that directly executes on the hardware.

In this chapter, the execution model of both systolic-based accelerators and Nvidia GPUs will be tackled, for Nvidia GPUs, warp-level execution will be tackled as a step deeper than what is discussed in Section 4.3.1 in which more involved timing analysis and architectural

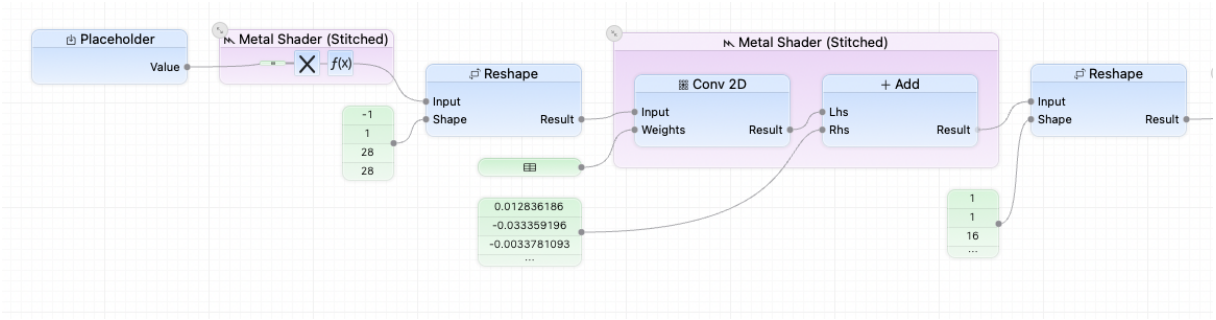


Figure 5.2: A section of an optimized pre-trained AI model for Apple Metal. Some operations are merged together.

nuances will be discussed. On the other hand, for systolic-based systems, focus will be directed towards understanding and analyzing the flow of data inside the processing elements (PEs).

1. This analysis is based on the description of AI inference operations only. Training requires a different set of hardware features that are not covered in this work.
2. Only execution of operations or instructions on Accelerators is analyzed, CPU-based execution pre- or post-execution is not taken into account.
3. Loading Time of a Model is not considered as it is a pre-execution step performed by the CPU and depends on the buses used to transfer the data.
4. The start of inference process assumes all data is already loaded into a DRAM buffer related to the accelerator, including Inputs, weights, biases, and operations. Due to the diverse nature of AI models and their tasks, the loading time of inputs is not taken into account since inputs could vary from text input, instructions, images, 3D models, etc. and the loading of the input data could vary significantly based on the system, the Input/Output (IO) hardware, and so on.

Section 5.2 will discuss the execution style and timing analysis of systolic-based accelerators, the section is split into two parts, Section 5.2.1 will discuss the execution model of systolic arrays and Section 5.2.2 will provide a method to calculate the number of Multiply and accumulate (MAC) operations required for an operation.

On the other hand, Section 5.3 will discuss Nvidia GPUs, where Section 5.3.1 will discuss the programming model of Nvidia CUDA, which is used to produce the operations of the model, Section 5.3.2 which describes the warp-level execution model which is required for Section 5.3.3 which provides an algorithm to estimate WCET of a kernel executing on an Nvidia GPU.

5.2 Timing Analysis of Systolic Arrays

Despite the differences in hardware implementations, most systolic accelerators utilize a set of processing elements (PEs) performing Multiply and Accumulate (MAC) operations, which are connected in a systolic assembly to stream data to from one PE to another. This reduces the time needed for evaluating matrix multiplication, vector multiplication, convolution, or other operations and consume significantly less energy due to the reuse of data. The systolic execution style and timing analysis will be discussed in this section.

5.2.1 Execution Model

A systolic array, which seems to be the de facto standard for spatial architectures, can be implemented in a variety of ways, but for the purpose of AI acceleration, the design of the systolic array relies heavily on the dataflow style of choice, described in Section 4.2.1 where the goal is to efficiently stream memory by reusing input data and weights multiple times after loading them from memory, which significantly reduces the needed bandwidth for memory-bound operations.

The output stationary dataflow is where the output of the operation is stored in the PEs themselves, and not streamed across the PEs. A typical PE with output stationary dataflow might look like Figure 5.3. Despite differences in implementation, Figure 5.3 represents the general view of an output stationary PE. The operating cycle of the PE can be described in an assumed **Load** and **execute** cycles, similar to other authors who propose a 2-stage pipeline [34]. The load cycle, shown in Figure 5.4 with the path highlighted in green, is where the weights are loaded vertically and activations are loaded horizontally from either the memory or from the previous PE. In the execute cycle, shown in Figure 5.5 with the path highlighted in green, the PE performs the MAC operation and stores the result in the PE itself in a special buffer which is then read in different operations after the execution is finished.

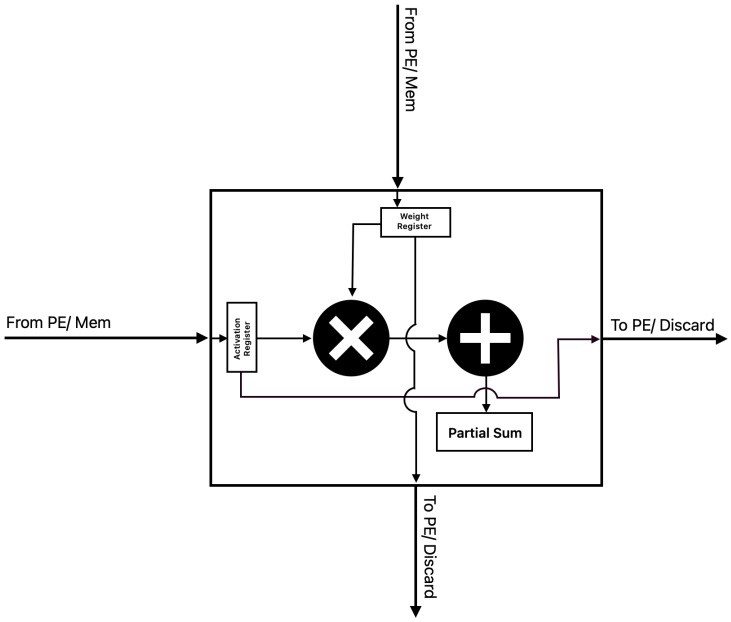


Figure 5.3: Assumed PE of a systolic array with output stationary dataflow.

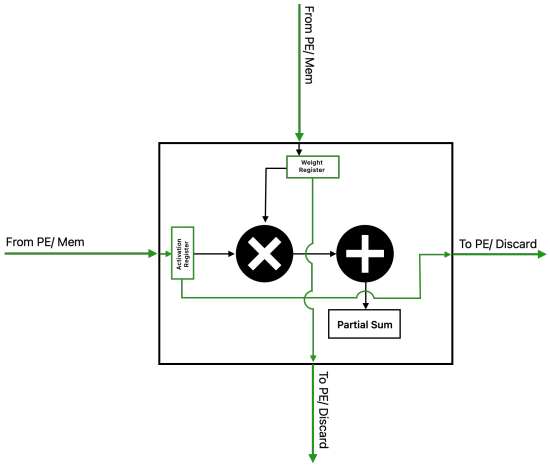


Figure 5.4: Assumed Load cycle of a PE in a systolic array with output stationary dataflow.

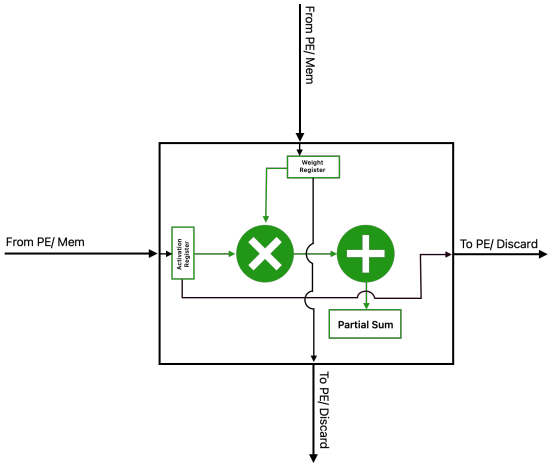


Figure 5.5: Assumed Execute cycle of a PE in a systolic array with output stationary dataflow.

A systolic array with a set of PEs perform matrix multiplication between matrix A of size 3×3 and matrix B of size 3×3 with row major indexing for the elements of both matrices where:

$$A_{3\times3} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} \times B_{3\times3} = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} \\ b_{1,0} & b_{1,1} & b_{1,2} \\ b_{2,0} & b_{2,1} & b_{2,2} \end{bmatrix} \tag{5.1}$$

produces the output matrix $C_{3,3}$ which is computed as follows in a systolic array with output stationary dataflow:

1. Load both matrices into respective buffers, typically in activations and weight buffers.
2. Dispatch input data from buffers to PEs with delays between each row of each matrix, represented by adding a number of delays (represented as zeros) at the beginning of each row.
3. Each PE receives a matrix column element from the PE or buffer above it and row element from the PE or buffer to the left of it, performs a MAC operation between the two elements, and stores the result in a partial-sum buffer inside the PE.
4. The PE passes the used column element to the PE below it and the used row element to the PE to the right of it.
5. Repeat step 3 until all the PEs have executed the MAC operation and the result is stored in the partial sum buffer.

An example of the execution within a systolic array is shown in Table 5.1, in which the PEs store the partial sums of the matrix multiplication inside the PE itself (output stationary dataflow). The result matrix $C_{3\times3}$ is equal in size and value to the partial sum buffers of each PE. The status of each PE is represented in colors which represents the systolic behavior of the PEs.

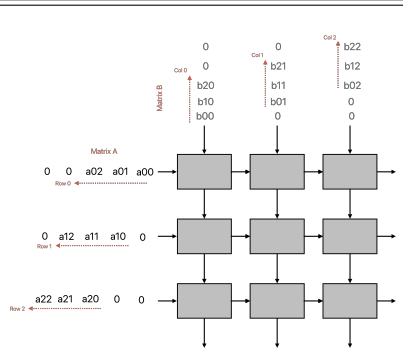
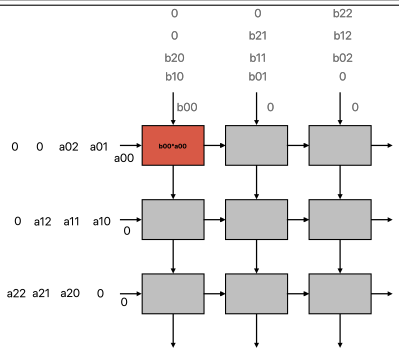
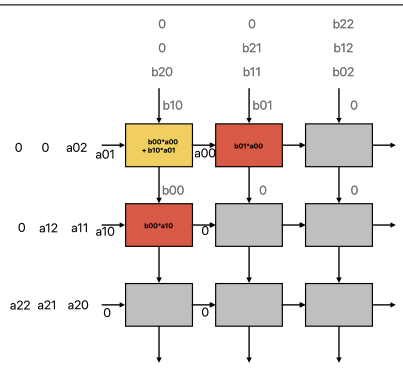
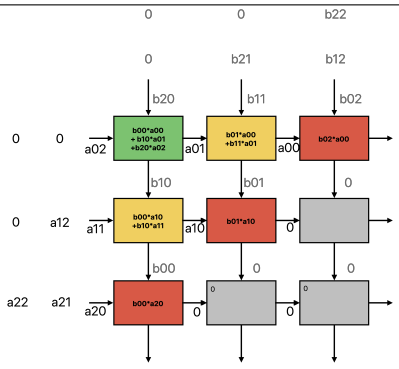
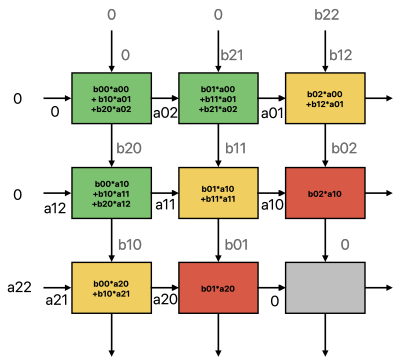
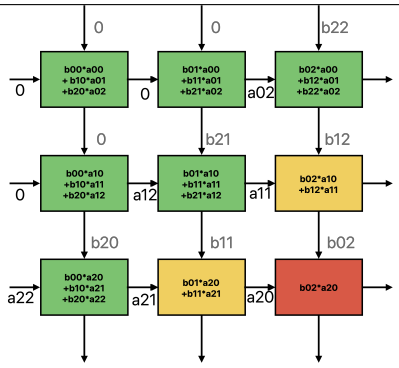
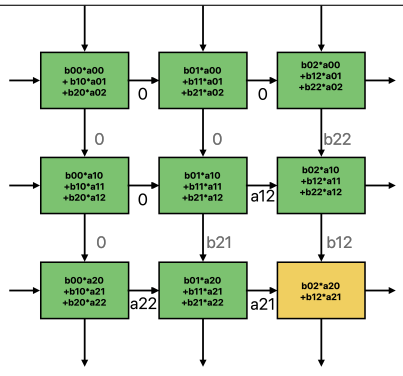
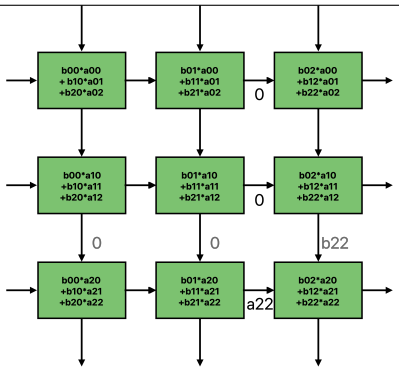
Step	Array	Step	Array
0		1	
2		3	
4		5	
6		7	

Table 5.1: Systolic Array with output stationary dataflow Example, grey: PE not active, red: $\frac{1}{3}$ of the sum accumulated, yellow: $\frac{2}{3}$ of the sum accumulated, Green: entire sum is accumulated.

Performing matrix multiplication between any two arbitrary matrices A of size $m \times k$ and B of size $k \times n$ on a single CPU core requires repeated memory access of the same data, which increases the need for higher bandwidth, and in total, the CPU has to perform $m \cdot n \cdot k$ multiplication operations and $m \cdot n \cdot k - 1$ addition operations sequentially.

For instance, $A_{3\times3} \times B_{3\times3}$ shown in Table 5.1 requires 27 multiplication operations and 18 addition operations when implemented serially. However, with the systolic array, each PE has to only perform 3 multiplication operations and 2 addition operations sequentially. The systolic behavior means at some points of execution, multiple PEs are active at the same time, for instance, step 3 and 4 in Table 5.1, 5 PEs are active, which is the maximum number for this example. Even though not all the PEs are active at the same time, the number of MAC operations needed to produce the matrix multiplication are less, in this example, 7 operations, compared to using a CPU core and without the overhead of utilizing multiple CPU cores.

Throughout a similar assembly in the previous example, a systolic array with a **weight stationary** or **input stationary** dataflow would store all the weights or inputs inside the PE and utilize only one input buffer for the activations and the output buffer for the results, which are the partial sums in this case. The weights or inputs are loaded into the PEs but do not move across the array throughout any cycle; instead, each partial sum is passed from each PE to the one below it, and the input data is passed from each PE to the one to the right of it. If there are no PEs below the current PE, the partial sum is stored in the output buffer.

Weight stationary dataflow is believed to be utilized in the Google TPU [35] as shown in Figure 5.6, where it shows that the weight buffer bandwidth is substantially less than that of the activations buffer (30GiB/s vs 167GiB/s). This implies that the weights are only loaded once from the buffer into the PEs for one operation, or one batch of one operation, whereas the activations need to stream continuously from the buffer to the PEs. Additionally, Figure 5.6 shows an Activation unit and a normalize/Pool unit, which implies that activation functions and pooling operations are not executed using the systolic array, but rather using a different execution unit in the TPU.

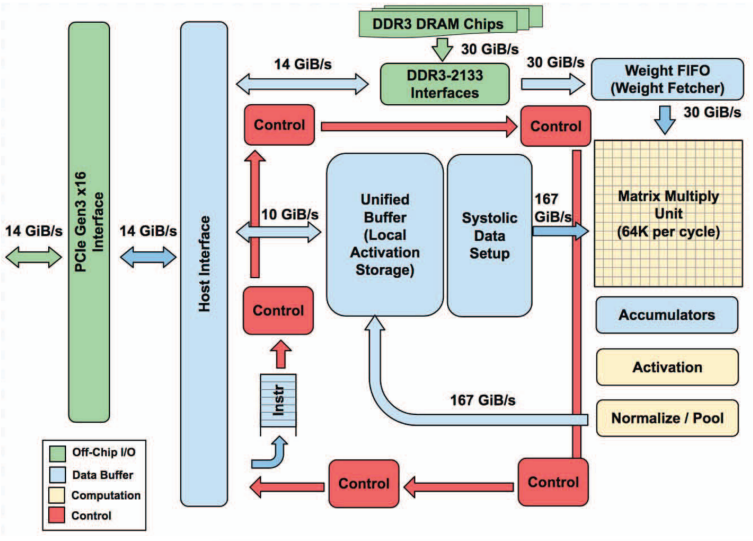


Figure 5.6: Google first-generation Tensor Processing Unit (TPU) [35].

One weight stationary PE can be visualized in Figure 5.7, which also can represent an input stationary PE. The weight register is loaded only once per matrix multiplication operation and used throughout, thus the name of weight stationary. The activations are streamed from the buffer to the PEs and across the PEs horizontally while the partial sums are stored in an external buffers. The activation matrix is loaded in a column major fashion for each row of PEs, and the output of each column of PEs represent each column of the output matrix. The load and execute cycles can be visualized in Figure 5.8 and Figure 5.9 respectively where the relevant paths are highlighted in green.

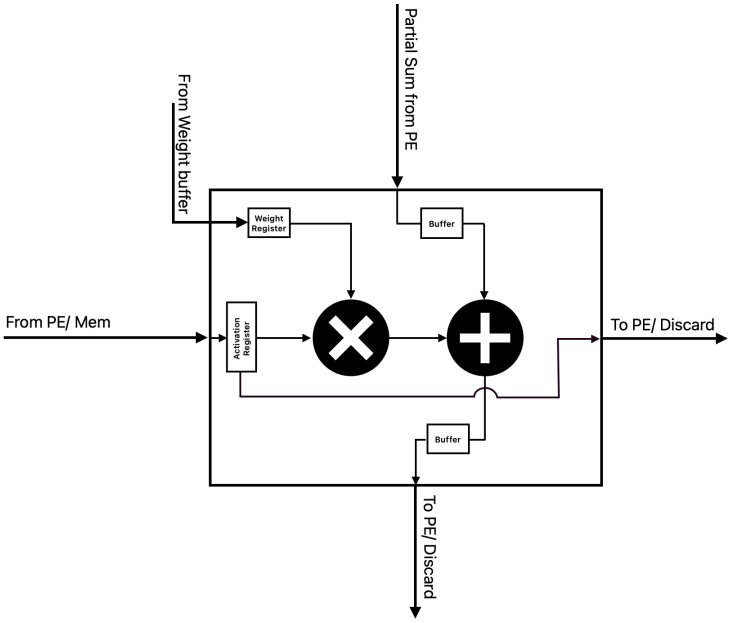


Figure 5.7: Assumed PE of a systolic array with weight stationary dataflow.

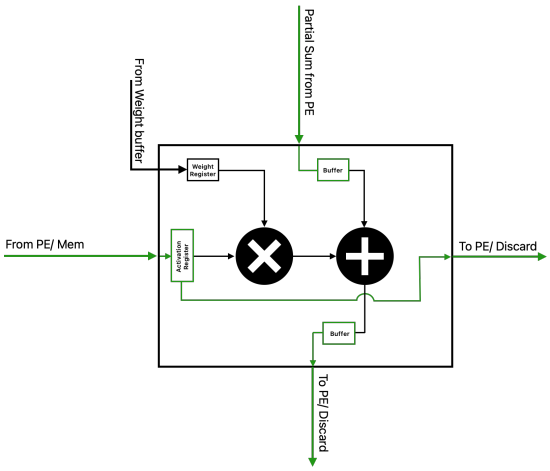


Figure 5.8: Assumed Load cycle of a PE in a systolic array with weight stationary dataflow.

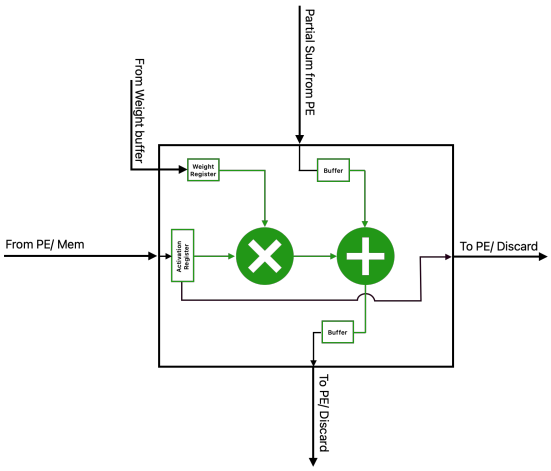


Figure 5.9: Assumed Execution cycle of a PE in a systolic array with output stationary dataflow.

The same example of performing $A_{3\times3} \times B_{3\times3}$ shown in Equation 5.1, is executed on a weight stationary systolic array and is visualized in Table 5.2, where the weights are preloaded into the PEs, the activations are streamed from memory to the PEs and across the PEs horizontally, and the partial sums are stored in an external buffers.

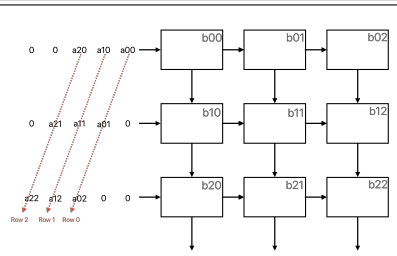
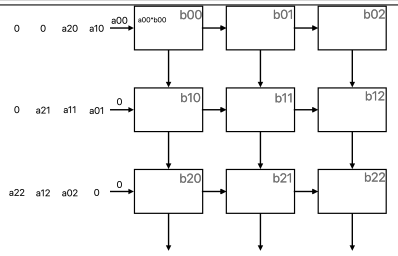
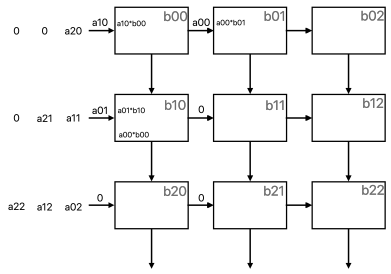
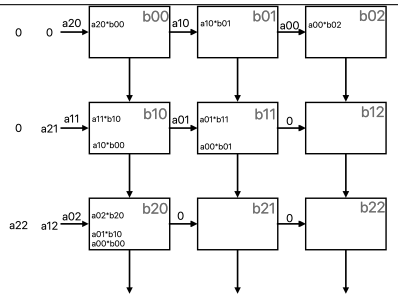
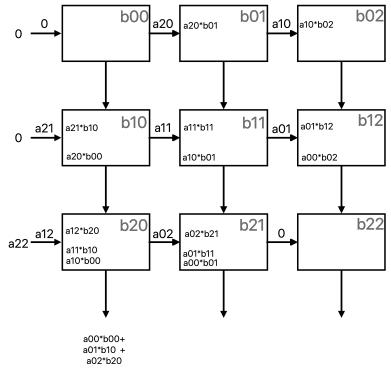
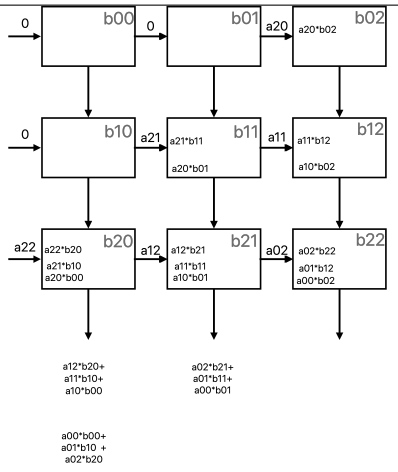
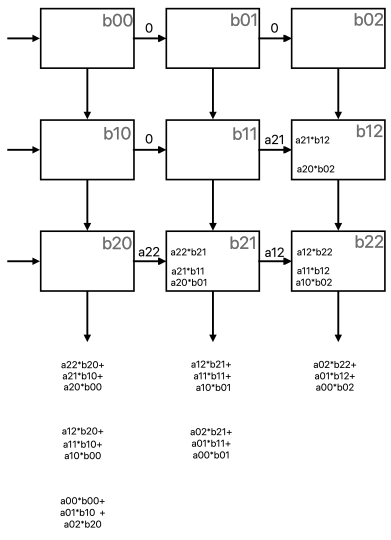
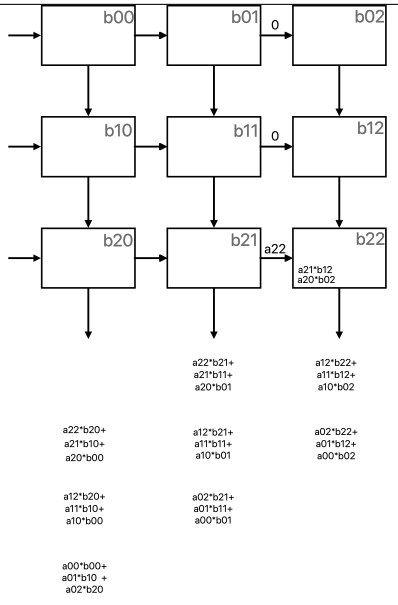
Step	Array	Step	Array
0		1	
2		3	
4		5	
6		7	

Table 5.2: Systolic Array with weight stationary dataflow Example.

Systolic arrays employ a set of PEs in a rhythmic fashion to perform MAC operations. The execution of a MAC operation in a systolic array is dependent on the shape of the array, and the number of MAC operations depends on the dataflow style used, the size of the matrices, and the number of batches. The number of MAC cycles used in an operation directly relates to the number of cycles as every MAC operation requires a certain number of cycles. However, due to proprietary nature, a lot of implementation data of state-of-the-art hardware is not publicly available. This timing analysis of systolic arrays is based on the public knowledge and some assumptions related to the specific cycles. For the purpose of this work, the number of MAC operations is studied, the results of the timing analysis can be then later scaled to accommodate the actual hardware implementation and used as a multiplier with cycles per MAC operation in future work.

For this purpose, the **Load** and **execute** cycles, in addition to all the delays associated, will be treated as one MAC operation regardless of how many actual cycles are required and will be studied in the next subsection.

5.2.2 Calculating Execution Time of an Operation

The regularity of the systolic array allows for simpler analysis of the execution time of a matrix multiplication process. The streaming process of matrix multiplication can be split into three parts, as shown in Figure 5.10, **Element streaming**, where all the elements of the matrix are streamed into the array, **Delay streaming**, where all the added delays are streamed into the array, and **propagation streaming**, where all the elements are streamed out of the systolic array.

Assuming a systolic array S of size $r_s \times c_s$ that performs a matrix operation without batching of matrix A of size $m \times k$ and a matrix B of size $k \times n$ resulting in an output matrix of the same size as the systolic array. The systolic array by design streams k elements from each row[1] from each matrix as shown in Equation 5.2.

$$\text{Element Streaming } S_E = k \tag{5.2}$$

Additionally, the systolic array introduces delays to the input data, where the first row and column have no delays, and each row has $r_s - 1$ delays and each column has $c_s - 1$ delays appended to the beginning. These delays are critical to the multiplication due to the systolic nature of the array. The delays are shown in Equation 5.3.

$$\text{Delay streaming } S_D = r_s - 1 \tag{5.3}$$

After the input data is loaded into the systolic array, the PEs start to perform the MAC operation and moving the data through the array, this also introduces a new set of streaming that propagates the input data through the array without any memory access. The number of operations needed to propagate the data through the array until the last element is in the last PE is shown in Equation 5.4.

[1] Or column, but this analysis will be done using row analysis.

$$\text{Propagation streaming } S_P = c_s - 1 \tag{5.4}$$

These operations are presented in Figure 5.10. The total number of MAC operations needed to perform the whole matrix multiplication is the sum of the MAC operations needed to load the data, shown in Equation 5.2, the delays shown in Equation 5.3, and the operations in Equation 5.4 to propagate the elements out of the array. This is shown as the total number of MAC operations for one matrix multiplication in Equation 5.5.

$$\begin{aligned} \text{Total MAC } Mac_{total} &= S_E + S_D + S_P \\ &= k + r_s - 1 + c_s - 1 \\ &= r_s + c_s + k - 2, \quad m \leq r_s, \quad n \leq c_s \end{aligned} \tag{5.5}$$

Where Equation 5.4 calculates to the total number of MAC operations required to perform one matrix multiplication where the resulting matrix is of equal size or smaller than the systolic array size.

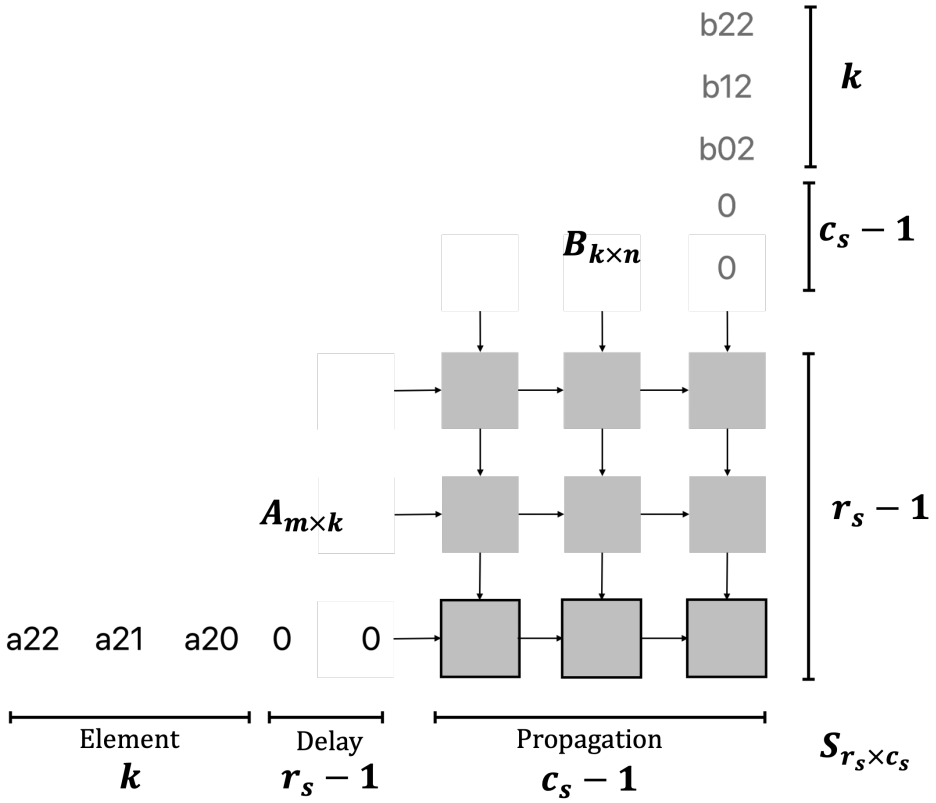


Figure 5.10: MAC operations needed for the execution of a matrix multiplication in an output stationary systolic array.

The whole operation of measuring the execution of matrix multiplication can be simplified as the study of the last row or column of the matrix. This is due to the repetitive nature of all the rows and columns moving into the systolic array in a repetitive manner. One may think of a window of size 1 at the first cycle going over the elements of the last row and gradually by 1 element every cycle until the window is of size c_s and to continue sliding over the elements until the last element is reached and is included in the window. This is

shown in Figure 5.11. At this point, all the elements of the last row or column are loaded and processed to the array, but in order to correctly compute the multiplication, they will need to propagate through the array, which will require additional $c_s - 1$ operations where the window starts to shrink by 1 element every cycle until the window is of size 1.

For instance, a matrix multiplication example shown in Equation 5.1 with two square matrices A and B of size 3×3 requires $3 + 3 + 3 - 2 = 7$ MAC operations, or 7 load and execute cycles, where the 7 load cycles depend on the memory bandwidth and the 7 execute cycles depend on how the MAC operation is implemented (i.e. one execute unit performs both operations or two separate cycles for each operation or multiple cycles per operation).

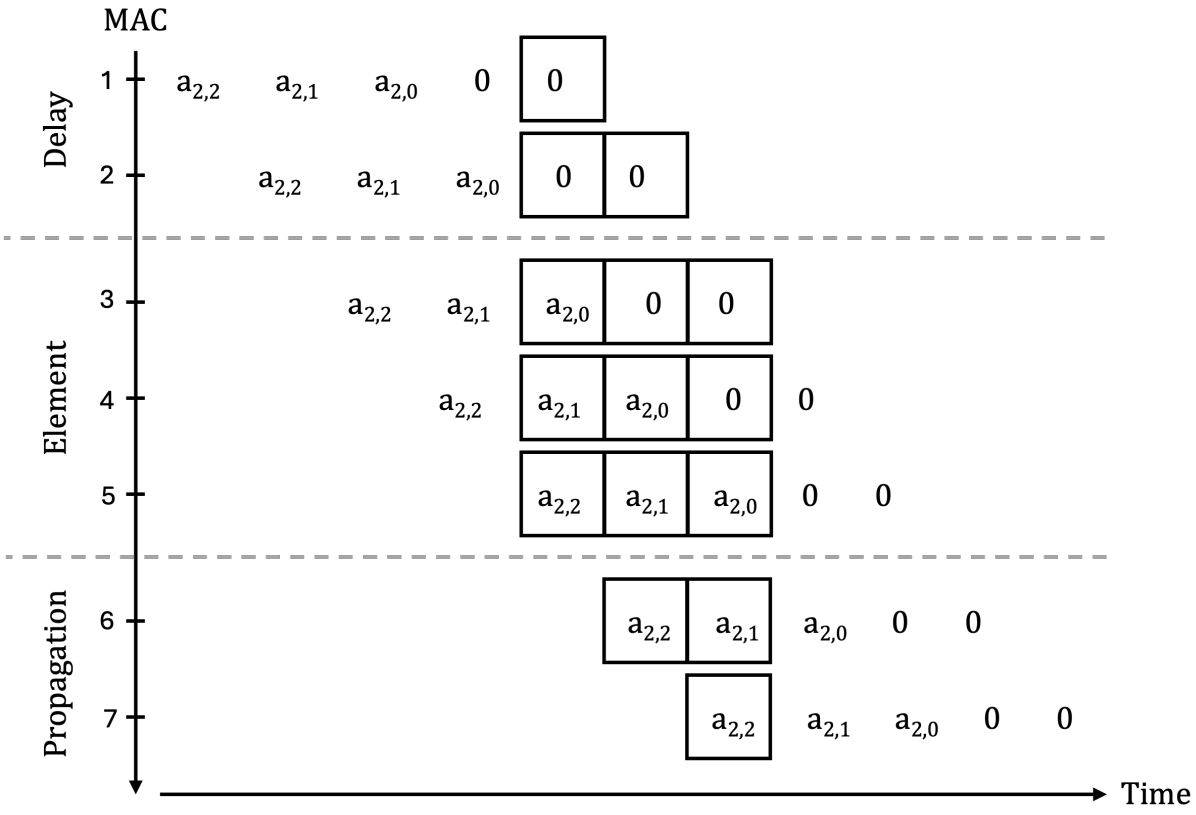


Figure 5.11: Processing of the last row of a matrix in a systolic array.

Since matrix multiplication depends on the size of the systolic array, the matrices $A_{m \times k}$ and $B_{k \times n}$ would extend the total number of cycles needed to perform the multiplication by how many batches or slices of each fit at once in a systolic array of size $r_s \times c_s$. This is calculated by finding the smallest integer that is larger than the number of row batches divided by m and the column batches divided by n, known as ceiling, this would produce the number of batches needed as a factor of multiplication for the overall number of MAC operations on one systolic array, as found by the authors of [36], extending the Equation 5.5 to:

For one systolic Array $S_{r_s \times c_s}$ and two matrices $A_{m \times k}$ and $B_{k \times n}$ of arbitrary size :

$$Mac_{total} = (r_s + c_s + k - 2)\lceil \frac{m}{r_s} \rceil \lceil \frac{n}{c_s} \rceil \tag{5.6}$$

Similarly, for input and weight stationary dataflows, the array would still stream the same number of matrix elements as described in Equation 5.2. The delay streaming described in Equation 5.3 and the propagation streaming described in 5.4 are also the same as in the output stationary dataflow. The total number of cycles needed to perform the multiplication is the sum of the operations needed to load the data, the added delays, and the operations needed to propagate the data through the array. Thus, for any systolic array S of size $s_r \times s_c$ performing matrix multiplication of matrix A of size $m \times k$ and matrix B of size $k \times n$ the number of MAC operation needed to perform this matrix multiplication is expressed in Equation 5.5.

5.3 Timing Analysis of Nvidia GPUs

In this section, the execution cycle of a CUDA kernel will be discussed in detail, identifying all the key factors that take place during the execution of a kernel that affect the number of cycles needed for execution and thus the worst-case execution time. This section will focus on the Nvidia Ampere architecture, specifically the A100 server GPU as the main target. This section will cover the programming model and some of the different paradigms associated with CUDA, the execution model of a kernel, and the development of the algorithm that statically estimates the WCET of a kernel running on an Ampere GPU.

5.3.1 Programming Model

Nvidia Describes the Compute Unified Device Architecture (CUDA) in [29] as a parallel computing platform and programming model that allows developers to use Nvidia GPUs for general-purpose processing. CUDA exposes a number of processing cores, in the order of thousands in general but different from one GPU to another, that perform scalar operations. These operations are programmed in a Kernel, which is a special kind of function that extends the C++ programming language, or through Parallel Thread Execution Virtual Instruction Set (PTX) and is executed on an Nvidia GPU when invoked. Similar to a typical function, a Kernel describes a set of sequential code that runs as a set of parallel threads.

CUDA organizes the threads (a kernel that was invoked to be executed) into a three-dimensional structure of threads, blocks, and grids. This is shown in Figure 5.12. Invoking a kernel takes the following form in the C++ extended syntax:

```
Kernel<<<numGrids, numBlocks, threadsPerBlock>>>(arguments);
```

Where the total number of threads to be executed is equal to $threadsPerBlock \times numBlocks \times numGrids$. The Execution of a code that contains a CUDA kernel is

shown in Figure 5.13, where when a kernel is invoked, the execution of kernel instructions is transferred to the Nvidia GPU.

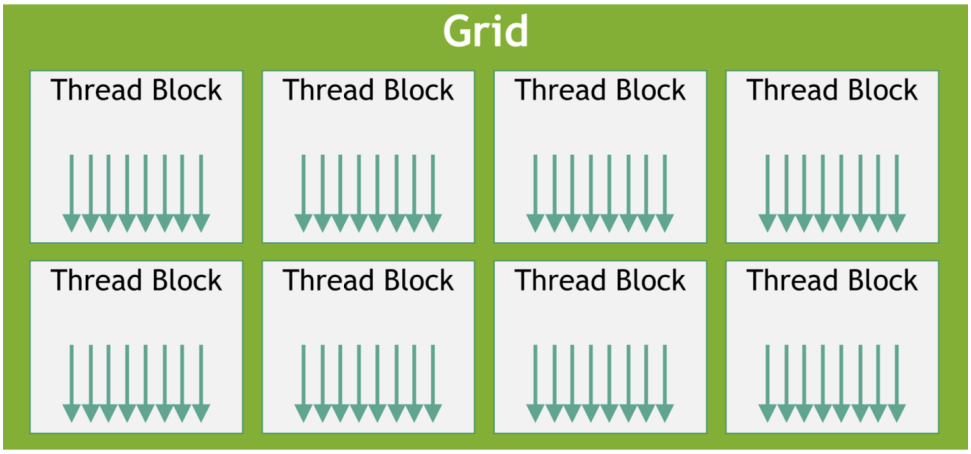


Figure 5.12: the organization of threads (Arrow shape), blocks, and Grids [37].

The threads are organized into blocks, where each block is executed by a single Streaming Multiprocessor (SM) within the GPU. These blocks are further grouped into grids, with each grid being executed by a single GPU or on a first-come, first-served basis. A maximum of 1024 threads can be accommodated within a block, each with its own private local memory and set of registers. In addition to these, blocks share memory accessible to all threads within the block, as well as global memory accessible to all threads across all blocks, and read-only constant and texture memory accessible to all threads across all blocks.

Individual threads are uniquely identified by their global index, which is a three-dimensional index that is exclusive to each thread within the grid. This index comprises a unique thread index, a block index, and a grid index. A unique one-dimensional global identifier can be calculated and utilized to access global memory on a thread-level granularity.

Kernels, along with the associated memory copy operations, are associated with streams, which are a sequence of operations that are executed in order. Streams are used to manage the execution of kernels and memory copy operations, and can be used to overlap the execution of multiple operations. Information about streams is described in Section 4.3.1.3.

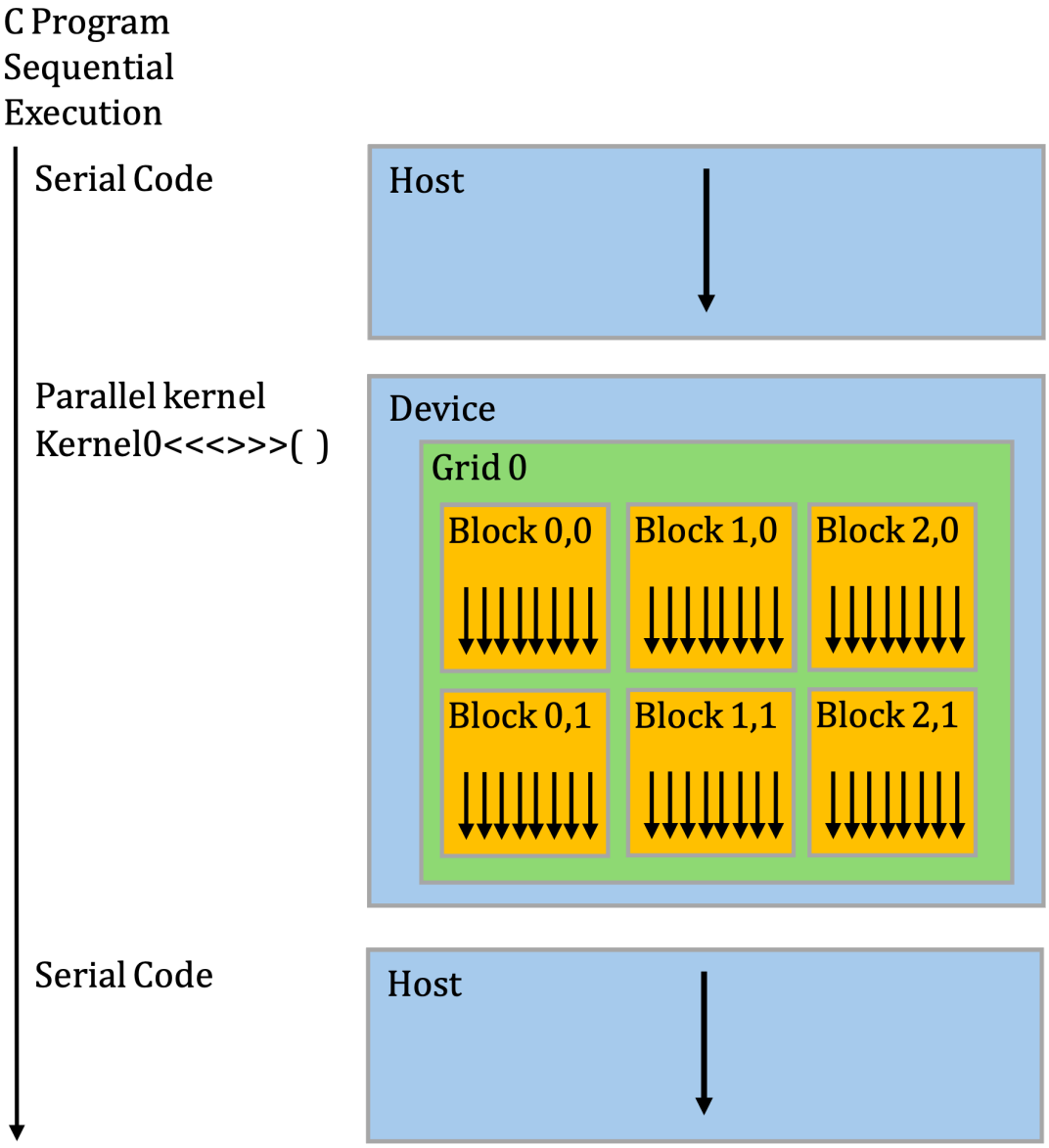


Figure 5.13: Code execution of CUDA Kernels, Host referes to CPU execution and Device refers to GPU execution [37].

A CUDA kernel, one such as shown in the following listing:

```
__global__ void add(int* a, int* b, int* c){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if ( i<N )c[i]=a[i]+b[i];
}
```

performs, as an example, addition operation on two vectors A and B and stores the results in Vector C. this kernel is intended to be executed N times, where N is the size of the vectors, and thus an Identifier is needed for each thread so that each thread can access one element in each vector. The general practice is to invoke as many threads as there are elements in the used data structure, in this case, the vectors A, B, and C. This CUDA C++ code is then compiled into PTX code, a Virtual instruction set that is not directly computed by the GPU cores, but acts as a middle layer between the CUDA C++ code and the machine code that is executed by the GPU. The PTX instructions produces for the above code is as follows:

```
 {
#Load a pointer to the location of vector a,b,c to regesters
ld.param.u64 %rd1, a;
ld.param.u64 %rd2, b;
ld.param.u64 %rd3, c;

#Load the thread indicies to regesters
mov.u32 %r2, %ctaid.x;
mov.u32 %r3, %ntid.x;
mov.u32 %r4, %tid.x;

# i = blockIdx.x * blockDim.x + threadIdx.x;
mad.lo.s32 %r1, %r3, %r2, %r4;

#Check if i < N
setp.gt.s32 %p1, %r1, 63;
@%p1 bra BB0_2;

#Main addition operation.
#Calculates the byte offset for index i.
mul.wide.s32 %rd5, %r1, sizeof(int);


cvta.to.global.u64 %rd4, %rd1;
add.s64 %rd6, %rd4, %rd5;  #Calculates the address of a[i].

cvta.to.global.u64 %rd7, %rd2;
add.s64 %rd8, %rd7, %rd5; #Calculates the address of b[i].

ld.global.u32 %r5, [%rd8]; #Loads b[i] to register %r5.
ld.global.u32 %r6, [%rd6]; #Loads a[i] to register %r6.
```

```
add.s32 %r7, %r5, %r6; # %r7 = %r5 + %r6.

cvta.to.global.u64 %rd9, %rd3;
add.s64 %rd10, %rd9, %rd5; #Calculates the address of c[i].
st.global.u32 [%rd10], %r7; #Stores %r7 in c[i].

BB0_2:
ret;
}
```

The PTX code is then compiled into Source and Assembly instructions (SASS) code, which is the machine code that is executed by the GPU. The SASS code for the above PTX code is as follows:

```
MOV R1, c[0x0][0x28]
S2R R6, SR_CTAID.X
HFMA2.MMA R7, -RZ, RZ, 0, 2.384185791015625e-07
ULDC.64 UR4, c[0x0][0x118]
S2R R3, SR_TID.X
IMAD R6, R6, c[0x0][0x0], R3
IMAD.WIDE.U32 R2, R6, R7, c[0x0][0x160]
IMAD.WIDE.U32 R4, R6, R7, c[0x0][0x168]
LDG.E R2, [R2.64]
LDG.E R5, [R4.64]
IMAD.WIDE.U32 R6, R6, R7, c[0x0][0x170]
IADD3 R9, R2, R5, RZ
STG.E [R6.64], R9
EXIT
BRA 0x71fca525c4e0
```

Whilst designing or invoking a kernel, a programmer is usually instructed to consider the theoretical occupancy of the GPU. Theoretical Occupancy of a kernel describes the ratio of active warps in a multiprocessor at a given time to the maximum number of active warps supported by the GPU. Theoretical Occupancy is a direct result of the number of threads per block, the amount of shared memory and registers used by the kernel, and the number of blocks per multiprocessor. The CUDA Occupancy Calculator, provided within the CUDA Toolkit, documents the total number of registers and the total amount of shared memory allocated for a block among other factors that can be statically calculated or considered by the programmer before the kernel is executed. This is documented by Nvidia in [38], where these static factors are provided in relation to the compute capability of the GPU, which is a number assigned by Nvidia that summarizes the hardware features of a GPU. For instance, the Nvidia A100 has a compute capability of 8.0, and Table 5.3 shows the occupancy limits for compute capabilities 8.0. For other compute capabilities, the limits are different and are documented in the CUDA Occupancy Calculator [38].

A kernel that is launched within the limits of the GPU's occupancy will be executed in one or fewer Waves. A wave is a ratio that represents the number of warps that run in parallel on one SM compared to the maximum number of warps that can run in parallel on one SM based on the restrictions described in Table 5.3. If there is at least one block

Parameter	Value
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	128
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	1024

Table 5.3: Occupancy limits for compute capabilities 8.0 [38].

that requires more resources than described in Table 5.3, the kernel will fail to launch.

5.3.2 Execution Model

An SM contains a set of warp schedulers. On the Nvidia A100, 4 warp schedulers exist per SM. Each warp scheduler is responsible for handling a subset of the warp-pool assigned to the SM, 64 warps per SM for the Nvidia A100 GPU. The warp schedulers issue one SASS instruction of one warp each clock cycle[38] to the pipeline associated with the type of instruction, including memory instructions. Thus, the execution of a kernel on instruction level is a function of not only thread-level parallelism but also instruction-level parallelism.

Thread level parallelism (TLP) is the number of threads that are executed in parallel, which is a function of kernel size and resource usage per thread and occupancy. One instruction is issued to all threads in a warp at the same time across used SMs, and each SM cycles through the available warps. Each thread in a warp has its own program counter. If threads within a warp diverge due to conditional paths, represented as branch to label SASS instruction, the warp executes one instruction from each branch to the threads following that branch, then switches to the next branch, and so forth, disabling threads not on that path. Branch divergence occurs exclusively within a warp; different warps execute independently, irrespective of whether they execute common or disjoint code paths. Since Nvidia Volta architecture, Independent Thread Scheduling is employed, enabling full concurrency between threads, irrespective of warp. This feature allows the GPU to maintain execution state per thread, including a program counter and call stack, and execute at a per-thread granularity. This means that individual threads in one warp can wait for data, diverge execution to different sub-branches independently of other threads in the warp, but they will still be grouped together for maximum efficiency (i.e. groups of threads executing one branch will be grouped together)[39, 37]. Execution of divergent paths in Independent Thread Scheduling is shown in Figure 5.14. Where

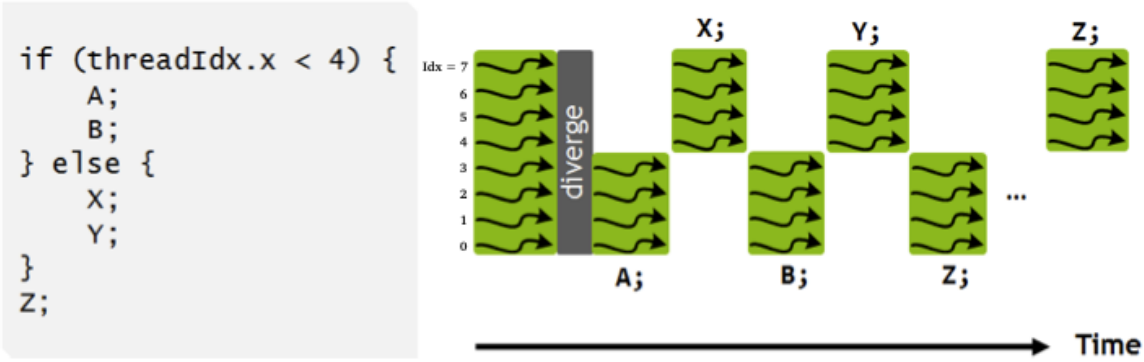


Figure 5.14: Execution of divergent path in Independent Thread Scheduling [39]. Threads following the same execution path will execute together while other threads are deactivated.

Divergence in a warp causes significant performance degradation. Each branch of the divergent path represents its own sub-warp within the one warp. Even though context switching between sub-warps is negligible, the warp scheduler will issue one instruction per sub-warp level, resulting in executing time equal to the number of sub-warps in one warp and their respective execution time.

Instruction level parallelism (ILP) deals more directly with how individual instructions are executed, each instruction issued by a warp scheduler is directly pushed into a pipeline,

and each instruction has its own latency which were documented to some level in [40, 41], which is the number of cycles it takes to execute the instruction. It was noticed that pipelines execute instructions in parallel, as in, a floating point set of instructions and an integer set of instructions can be executed in parallel as they use different pipelines. Nvidia Nsight Compute [38] documents the different pipelines and some of which are described as below:

- Arithmetic Logic Unit (alu): responsible for execution of most bit manipulation and logic instructions, integer instructions (excluding IMAD and IMUL), and fast FP32 to FP16 conversion.
- Convergence Barrier Unit (cbu): Processes with warp-level barriers, branching, and convergence.
- Fused Multiply Add/Accumulate (fma): Queues 32-bit floating-point operations in addition to integer multiplication. It is split intoo **fmaheavy** and **fmalite** pipelines.
- Fused Multiply Add/Accumulate Heavy (fmaheavy): performs FP32 arithmetic (FADD, FMUL, FMAD), FP16 arithmetic (HADD2, HMUL2, HFMA2), integer multiplication operations (IMUL, IMAD), and integer dot products.
- Fused Multiply Add/Accumulate Lite (fmalite): performs FP32 arithmetic (FADD, FMUL, FMA) and FP16 arithmetic (HADD2, HMUL2, HFMA2).
- Half-precision floating-point (fp16): performs paired FP16 instructions (FP16x2) and fast FP32-to-FP16 and FP16-to-FP32 converter
- Double-precision floating-point (fp64): Not described, hardware-implementation dependent.
- load and store (lsu): Queues memory load and store instructions for global, local, and shared memory.
- Tensor Core (tc): executes UTCBAR, UTCCP, UTC*MMA, UTCSHIFT and UTC*SWS instructions.
- Tensor: executes various MMA instructions.
- Transcendental and Data Type Conversion Unit (xu): Queues special functions such as sin, cos, and reciprocal square root in addition to float-to-int and int-to-float conversions.

During experimentation, it was found that the execution time heavily depends on the most-used pipeline. For instance, if the FMA pipeline is the most used, the execution time will be heavily dependent on the FMA instructions used even if other pipelines are used in parallel. This matches the latency hiding techniques used in Nvidia GPUs, where the GPU will execute instructions from different pipelines in parallel to hide the latency of the most used pipeline. As far as this work goes, there wasn't a well-defined documentation mapping certain instructions to certain pipelines; however, Nvidia provides a list of SASS instructions and descriptions in [42] and the previously mentioned pipelines in [38] through which some form of mapping can be done.

Estimating WCET for a kernel depends on the following factors:

- The scheduling of a kernel in a stream and the priority of the stream as described in Section 4.3.1.
- TLP: The number of threads in a block, and blocks in a grid, and the resulting number of warps per SM.
- ILP: The number of instructions per pipeline, and how balanced the pipelines are.
- Memory Access patterns of Global, shared, and Constant memory.

The relationship between WCET factors and AI models can be described through a three-dimensional relationship between the CUDA execution model and the AI model parameters as shown in Figure 5.15. An AI model consists of a sequence of layers describing operations such as convolution, matrix multiplication, etc., that need to be executed sequentially as the input of one layer is the output of the previous one. The number of operations in a model is directly related to the number of CUDA kernels invoked. Each CUDA kernel implementation processes each element of the input of each layer in parallel, for example, pixels in an image, where each layer requires an input of a certain size, i.e. the dimensions of an image and the number of channels. One layer can be implemented in one CUDA kernel or multiple, but the invocation of the kernel, or operation, when it comes to the number of blocks and threads, is a function of the input size. Additionally, the pipelines utilized in a kernel are a function of the quantization of the model. Quantization is the process of converting the weights and biases of a model from a larger representation to a smaller one, for instance, from 32-bit floating point to 16-bit floating point or 8-bit integer. Quantization impacts the accuracy of a model due to loss of precision, but certain GPUs implement more efficient pipelines for certain quantization levels and can be noticed in the delay of instructions associated with that pipeline.

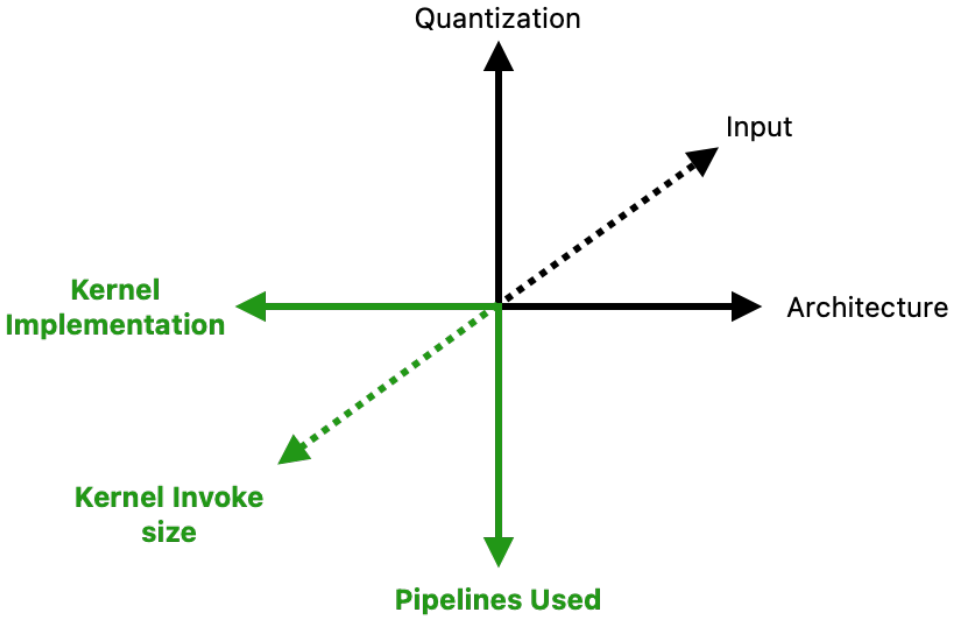


Figure 5.15: AI model parameters (Black) in relation to CUDA kernel execution (Green).

5.3.3 Algorithm for WCET Estimation

The following algorithm is proposed to estimate the WCET of a CUDA kernel. The algorithm is based on the factors described in Section 5.3.2 and the relationship between AI models and CUDA kernels described in Section 5.3.2. This algorithm deals directly with the CUDA kernels used in the AI model executing on one GPU, and not the AI model as a whole. As the AI model is a sequence of operations that are executed in parallel, the execution time of the model is the sum of the execution time of each operation. To make the algorithms more readable, the used notations are described in Table 5.4.

Notation	Description
$\mathcal{I}$	List of SASS instructions
Size	$\{Blks, Thrds\}$ (block and thread count. Assumes one GPU-inference only)
$Cycles$	Accumulated cycle count
W	Assigned/Active Warps per SM: $\frac{\text{Size.Blks} \times \text{Size.Thrds}}{32 \times N_{SM}}$
$\mathcal{P}$	Set of used pipelines
$\lvert\mathcal{P}[p]\rvert$	Number of instructions in pipeline p
d_{global}	Global memory access delay
d_{shared}	Shared memory access delay
d_{L1}	L1 cache access delay
d_{register}	Register access delay (negligible, part of instruction delay)
$\mathcal{A}_{\text{global}}$	Set of previously loaded global addresses
$\mathcal{A}_{\text{const}}$	Set of previously loaded constant addresses
$GetMemoryDelay(Inst)$	Algorithm that returns the memory access delay of instruction $Inst$
$GetPipeline(Inst)$	Look-up table of pipeline and associated instructions $Inst$
$GetDelay(Inst)$	Look-up table of instruction and associated delays $Inst$

Table 5.4: Notation Summary for WCET Algorithm.

The Worst Case Execution Time Algorithm 1 takes a list of SASS instructions $\mathcal{I}$ and the size of the kernel in terms of blocks and threads as an input, and returns the number of cycles needed for the kernel to execute. The algorithm first calculates the number of active warps per SM, then iterates through the list of instructions, categorizing them into memory access instructions and data processing instructions. The algorithm then calculates the number of pipelines used, the average number of instructions per pipeline, and the maximum number of instructions in a pipeline including latency hiding in a saturated pipeline that has α times the instructions as the other pipelines, where α is a multiplier which is left to the analyzer to choose. The algorithm then calculates the number of cycles needed for the kernel to execute based on the number of instructions

in each pipeline and the number of active warps per SM. The algorithm is based on the factors described in Section 5.3.2 and the relationship between AI models and CUDA kernels described in Subsection 5.3.2.

Algorithm 1 Worst-Case Execution Time (WCET) Estimation

Input: $\mathcal{I} = [\text{Inst}_1, \text{Inst}_2, \ldots, \text{Inst}_n]$, Size $= \{Blks, Thrds\}$
Output: $Cycles_{WCET}$
Intermediate Variables: $\mathcal{P} = \{P_0, P_1, \ldots, P_m\}$, $Cycles \leftarrow 0$
$W \leftarrow \frac{\text{Size.Blks} \times \text{Size.Thrds}}{32 \times N_{SM}}$
for $Inst \in \mathcal{I}$ **do**
    $p \leftarrow \text{GetPipeline}(Inst)$
    if $Inst \in \{Inst_{Mem}, Inst_{combined}\}$ **then**
        $Cycles \leftarrow Cycles + \text{GetMemoryDelay}(Inst)$
        if $Inst \in Inst_{combined}$ **then**
            $\mathcal{P}[p] \leftarrow \mathcal{P}[p] \cup \{Inst\}$
        end if
    else if $Inst \in Inst_{Data}$ **then**
        $\mathcal{P}[p] \leftarrow \mathcal{P}[p] \cup \{Inst\}$
    end if
end for
$N \leftarrow |\mathcal{P}|$ ▷ Number of pipelines
$MaxPipelineSize \leftarrow \max_p(|\mathcal{P}[p]|)$
$TotalInstructions \leftarrow \sum_p |\mathcal{P}[p]|$
$AvgPipelineSize \leftarrow \frac{TotalInstructions}{N}$
if $\exists p$ such that $|\mathcal{P}[p]| > \alpha \times AvgPipelineSize$ **then**
    $Cycles \leftarrow Cycles + \sum_{\text{Inst} \in \mathcal{P}[p]} \text{GetDelay}(Inst) \times W$
else
    $Cycles \leftarrow Cycles + \sum_{p \in \mathcal{P}} \sum_{\text{Inst} \in \mathcal{P}[p]} \text{GetDelay}(Inst) \times W$
end if
$Cycles_{WCET} \leftarrow Cycles$

The WCET estimate that is provided from Algorithm 1 relies on the static nature of AI models, as models process inputs of a constant size and produce outputs of predetermined labels or tokens through the application of a set of operations. Algorithm 1 assumes no branching in execution; However, branching can still occur in some cases. To deal with these cases, the person or any future implementation performing the analysis must split the kernel between branching and convergence points and treat each branch, even if that path was taken once, as its own kernel. Kernels experiencing a different path will be disabled, while others on the same path will execute together. This means the same calculated active warps number will be used for all paths taken, even if only one thread in a warp is taking that path.

To calculate Memory Delays, Algorithm 2 is proposed based on the resources described in [29, 38]. The algorithm takes one SASS instruction as input and returns the number of cycles needed for the instruction to execute. The algorithm categorizes the instruction into global, shared, and constant memory access instructions, then calculates the number

of cycles needed for the instruction to execute based on the memory access pattern of the instruction. The algorithm also keeps track of previously loaded addresses to take into account L1 cache hits. Global load instructions load one 128-byte patch to L1 Cache; an address might be already loaded to L1 even though it was not already loaded before.

Algorithm 2 Memory Access Delay Estimation

Input: $\mathcal{I}$ ▷ Instruction
Output: $Cycles \leftarrow 0$
Intermediate Variables: $\mathcal{A}_{\text{const}}, \mathcal{A}_{\text{global}}$ ▷ Previously loaded addresses
if $\mathcal{I} \in \text{Mem}_{\text{global}}$ **then**
    if Kernel is streaming **then**
        $Cycles \leftarrow Cycles + d_{\text{global}}$
    else if $\mathcal{I}$.address $\notin \mathcal{A}_{\text{global}}$ **then**
        $Cycles \leftarrow Cycles + d_{\text{global}}$
        $\mathcal{A}_{\text{global}} \leftarrow \mathcal{A}_{\text{global}} \cup \{\mathcal{I}.\text{address}\}$
    else if $adress \in Read - Only$ **then**
        $Cycles \leftarrow Cycles + d_{\text{L1}}$
    else
        $Cycles \leftarrow Cycles + d_{\text{L1}}$
    end if
else if $\mathcal{I} \in \text{Mem}_{\text{shared}}$ **then**
    $Cycles \leftarrow Cycles + d_{\text{shared}}$
else if $\mathcal{I} \in \text{Mem}_{\text{constant}}$ **then**
    if $\mathcal{I}$.address $\notin \mathcal{A}_{\text{const}}$ **then**
        $Cycles \leftarrow Cycles + d_{\text{global}}$
        $\mathcal{A}_{\text{const}} \leftarrow \mathcal{A}_{\text{const}} \cup \{\mathcal{I}.\text{address}\}$
    else if DynamicBankAccess $\vee$ DynamicOffsetAccess **then**
        $Cycles \leftarrow Cycles + d_{\text{global}}$
    else
        $Cycles \leftarrow Cycles + d_{\text{register}}$
    end if
end if

Memory access patterns in Algorithm 2 for Nvidia A100 can be described in the following set of rules:

- Global memory variables are loaded as a 128-byte coalesced value at once to the L1 cache despite the size of variable used in the kernel [29, 38].
- variables previously loaded by the kernel from Global memory are always in L1.
- Read-only Data for the entire lifetime of the kernel, i.e. variables identified with const and restrict are always cached [29, 38].
- Data that is not read-only for the entire lifetime of the kernel cannot be cached, before first-load, in the unified L1/texture cache [29, 38].
- Global memory reads can be cached to L1 cache if some compiler-flags were used.
- Global Read/write for A100 requires 290 cycles [41].

- Shared memory access requires 23 cycles for read, and 19 cycles for write [41].
- Constant memory requires the same cycles as Global memory [29, 38].
- previously loaded constant memory address or one within 128 bytes of a previously loaded address is treated as register delay [29, 38], substituted with 0 additional delay.

Shared Memory is much faster than global memory, however, it needs to be loaded by the kernel during runtime. Constant Memory is not modifiable by the kernels, only by host code and unless they were accessed dynamically, the performance is as fast as the register file.

6 Evaluation

This chapter applies the developed algorithms for measuring WCET of kernels on an Nvidia GPU. For other AI accelerators using systolic arrays, the lack of documentation for the individual accelerators and the undocumented nuances of implementation prevents systematic evaluation of the accelerators. Additionally, the found equations match the results of previous works such as that of [36, 23]. This chapter focuses on the Nvidia A100 GPU, describes the setup of the experimentation, applies the WCET algorithm to a set of examples taken directly from an AI model, discusses the execution of some instructions, and provides a comparison between the estimated WCET and the measured average execution time.

6.1 Methodology

In this section, the methodology of assessment of Algorithm 1 is described in detail before performing the actual experiments. The Aim of this work is to estimate the WCET, however, this estimation is carried out in cycles of execution, not in the time domain, therefor, an accurate way of measuring the cycles of execution that is independent of measuring time is substantial to the assessment, for this reason, Nvidia Nsight Compute [38] is used to profile the kernel in question. Nvidia Nsight Compute is an interactive profiler of CUDA executables that directly collects execution metrics to help developers optimize their CUDA implementations by identifying and visualizing bottlenecks of execution. For this use case, the one most important metric is the execution cycle of the kernel. As far as we are aware, this is the most accurate way of getting the execution cycles of a kernel.

The methodology of testing the WCET Algorithm 1 can be described as follows:

- **Step 1.** Compile a C-language code that executes a kernel on the GPU with no optimization and with debugging flags. Optimization might break consistency between higher-level code and the Instructions used, additionally, it may merge or change paths, type of instructions used, etc., to optimize performance which is not necessary for this work.
- **Step 2.** Extract the SASS instructions from the executable, multiple tools can do that, one way is to use Nvidia Nsight Compute, another is to use Nvidia's CUDA object dump (cuobjdump) command line tool.
- **Step 3.** Estimate the WCET of the kernel using the SASS instructions and launch parameters.
- **Step 4.** Profile the Kernel by attaching the compiled executable to Nvidia Nsight Compute, which will run each kernel 42 times and provide the average execution cycles for each kernel in the executable.

- **Step 5.** Compare the actual obtained execution cycles and the estimated WCET.

The process of Estimating using the WCET algorithm is done manually at this point and requires human intervention, as opposed to an automated script that implements the WCET algorithm.

6.2 Experimentation Setup

The setup used to carry out this analysis consists of a Personal Computer running Ubuntu 22.04 (Jammy Jellyfish), Host side specifications are as follows:

- Processor: X86-64 Intel i5-13500 Central Processing Unit.
- Memory: 64 Gigabytes(GB) of Double Data Rate 4 (DDR4) Random Access Memory (RAM).
- CUDA Version: 12.4.
- Nsight Compute version: 2023.1.0.0.

Device specifications of the used GPU are as follows:

- Name: Nvidia A100.
- Connection type: Peripheral Component Interconnect Express (PCIE).
- Memory: 40GB Graphics DDR6 (GDDR6) RAM.
- Clock Rate: 1410000 Hertz.
- Streaming Multiprocessors: 108.

This setup is used to analyze one inference operation at a time, where all the resources of the Nvidia A100 will be available for each of the operations in the model. Multiple inference operations execution at the same time is possible but requires the utilization of the scheduling rules and methodologies described in Section 4.3.1 which are not implemented in this model and will not be covered in this work.

6.3 Experiments

In this chapter, the methodology described previously will be applied to a set of examples of different kernel sizes, instruction-wise, to assess the algorithm. In this section, a simple detailed example will be shown in depth in addition to multiple AI-based kernels.

6.3.1 Introductory Example

For this example, a vector-adding kernel will be used as an example. This kernel is designed to occupy 100% of the theoretical capacity of the A100 GPU. One can immediately notice the streaming behavior of this kernel, where data is loaded directly from the global memory based on the global address of the kernel and never reused again.

```
1    #define threadsPerBlk 1024
2    #define blksPerGrid 108*2
3    #define N threadsPerBlk*blksPerGrid
4
5    __global__ void add(int* a, int* b, int* c)
6    {
7        int i = blockIdx.x * blockDim.x + threadIdx.x;
8        c[i]=a[i]+b[i];
9    }
```

This kernel was invoked and analyzed with Nvidia Nsight Tool, and the tool reported an **average of 5145 cycles of execution**, initial memory copy before and after the operation is not included. The SASS instructions used in this kernel are presented in Table 6.1 alongside the sources and destinations of the instructions, which can help further determine embedded operations within one instruction, such as embedded memory loads, etc.

	Instruction	**Dest**	**Src**	**Src**	**Src**	**Src**	**Pipeline**
1	MOV	R1	c[0x0][0x28]				const
2	S2R	R6	SR_CTAID.X				LSU
3	HFMA2.MMA	R7	Rz	RZ	0	2.38E-07	TC
4	ULDC.64	UR4	c[0x0][0x118]				const
5	S2R	R3	SR_TID.X				LSU
6	IMAD	R6	R6	c[0x0][0x0]	R3		fma + const
7	IMAD.WIDE.U32	R2	R6	R7	c[0x0][0x160]		fma + const
8	IMAD.WIDE.U32	R4	R6	R7	c[0x0][0x168]		fma + const
9	LDG.E	R2	[R2.64]				LSU
10	LDG.E	R5	[R4.64]				LSU
11	IMAD.WIDE.U32	R6	R6	R7	c[0x0][0x170]		fma + const
12	IADD3	R9	R2	R5	RZ		ALU
13	STG.E	[R6.64]	R9				LSU

Table 6.1: Vector addition kernel instructions.

Most of the instruction in Table 6.1 perform the global index (i) calculation, then loading the values of the a[i] and b[i] from global memory, adding the two values, and storing the result in the address of c[i]. Now, one can analyze the delays of each instruction with Algorithm 1 with $W = 64$ based on the kernel invoke size. from Table 6.1, one can see that no pipeline of execution is saturated, and thus all instruction delays need to be taken into account. Based on the instruction delays described in [41], and the WCET Algorithm 1, the WCET analysis is carried out and described in Table 6.2.

The Delay of the first "mov" instruction is unclear, and is assumed as a register move instruction merged with a constant memory load of a static unique address. For worst case estimation, the latency of constant memory access is equal to global memory access in this case. Other instructions such as number 6,7,8, and 11 also load from unique static constant memory addresses in addition to performing a specific operation. According to Algorithm 2, a unique static address access of constant memory will require the same

	Instruction	Multiplier	delay	total cycles
1	MOV	1	290	290
2	S2R	W	10	640
3	HFMA2.MMA	W	4	256
4	ULDC.64	1	290	290
5	S2R	W	10	640
6	IMAD	1	290	290
		W	4	256
7	IMAD	1	290	290
		W	4	256
8	IMAD	1	290	290
		W	4	256
9	LDG.E	1	290	290
10	LDG.E	1	290	290
11	IMAD	1	290	290
		W	4	256
12	IADD3	W	2	128
13	STG	1	290	290
				5298

Table 6.2: WCET Analysis of vector addition kernel.

delay as global memory access. For global memory load instructions, all loads are treated as unbuffered due to the absence of read-only signifiers for global memory variables, and no compilation flags were set to force global memory caching, and the streaming nature of the kernel (no memory address is reused at any point)

The reported **average execution cycles is 5145 cycles**, whereas the **calculated WCET is 5298 cycles** based on the analysis in Table 6.2 .

6.3.2 AI Model Examples

In this subsection, the same WCET analysis will be applied to an actual AI model designed to classify whether an image depicts a sugar beet plant or a weed in real time for optimized agricultural activities as part of 5G-AGRICULTURE-ML project [43]. The AI model is in PyTorch format and loaded using Python code, available in the Appendix, that utilizes CUDA for inference. The Python code was executed using Nvidia Nsight Compute to analyze the kernels used in the inference process. The AI model structure is shown in Figure 6.1, where a 32x32 image of 3 channels is loaded and processed by a set of operations, shown as circles for scalar operations and squares otherwise.

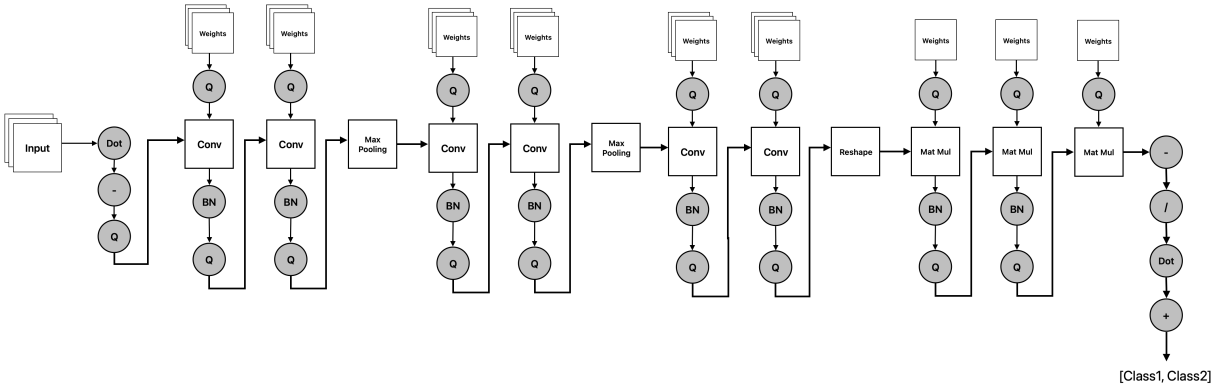


Figure 6.1: Sugar beet detection model, q: Quantization, Dot: Scalar product, -: Scalar Subtraction, +: scalar addition, /: scalar division, BN: Batch Normalization, Conv: Convolution, Mat Mul: Matrix Multiplication.

Nvidia Nsight Compute reported a total of 181 kernels within for the entire model, most of which are named element-wise and vectorized element-wise kernel, but notably, some kernels are named Implicit convolve, bn, and dot kernel, implying convolution operation kernel, batch normalization kernel, and matrix product kernel respectively. It is Recognizable that kernel implementation is not one-to-one mappable to the model due to the nature of implementation of the CUDA binaries for the PyTorch library, as they are pre-compiled at distribution or installation of the library. However, this does not impact the WCET analysis since it is applicable directly on the kernel level not operation level. Some select kernels will be analyzed in this section.

6.3.2.1 Vectorized Element-wise Kernel

The first kernel implemented in this model is labeled as Vectorized Element-wise kernel, it is a comparatively small kernel based on the reported instructions. Unfortunately, no source-code of the kernel is available, but, it is relatively easy to understand in the context of this model. This kernel was invoked with 6 blocks, each has 128 threads. The SASS instructions of the kernel are as follows:

```
1        IMAD.MOV.U32 R1, RZ, RZ, c[0x0][0x28]
2        S2R R0, SR_CTAID.X
3        ULDC.64 UR4, c[0x0][0x118]
4        SHF.L.U32 R0, R0, 0x9, RZ
5        IADD3 R3, -R0, c[0x0][0x160], RZ
6        ISETP.GE.AND P0, PT, R3, 0x200, PT
7  @!P0  BRA 0x7351b325c540
8        S2R R7, SR_TID.X
9        IMAD.MOV.U32 R5, RZ, RZ, 0x4
10       IMAD.WIDE R2, R0, R5, c[0x0][0x178]
11       IMAD.WIDE.U32 R2, R7, 0x10, R2
12       LDG.E.128 R8, [R2.64]
13       IMAD.WIDE R4, R0, R5, c[0x0][0x170]
14       IMAD.WIDE R4, R7, 0x10, R4
15       FMUL R11, R11, c[0x0][0x168]
```

```
16      FMUL R10, R10, c[0x0][0x168]
17      FMUL R9, R9, c[0x0][0x168]
18      FMUL R8, R8, c[0x0][0x168]
19      STG.E.128 [R4.64], R8
20      EXIT
```

This kernel calculates the global thread index, loads 128 bits (16 bytes, or 4 floating-point values) of global memory, splits the 128 bits into four floating-point values, multiplies each by another floating-point value loaded from memory, combines the 4 floating points into one 128 bits store instruction. This matches the first scalar dot-product operation in Figure 6.1, the input of 32 width $\times$ 32 height $\times$ 3 channels $= 3072$ pixels is processed by 6 blocks $\times$ 128 threads/block $= 768$ threads each processing 4 floating-point values, resulting in 768 threads $\times$ 4 pixel/thread $= 3072$ pixels

Now to analyze the WCET, the value of active warps is calculated as $W = \frac{6 \times 128}{32 \times 1} = 24$, since the number of threads is less than the maximum threads executable on one SM of the A100 (768 out of a maximum of 2048), this will execute only on one SM in 24 warps. Table 6.3 shows the pipelines and the delays of the instructions for the WCET analysis. This kernel shows an example of data reuse, where instructions number 15 - 18 use the same constant memory address, in which the first load was in instruction number 15, and all the subsequent instructions use a cached value of the constant address which Nvidia claims to be as fast as register access. The branch instruction (instruction number 7) is used as safeguard to exit execution if the global address is larger than a certain number. Despite the fma pipeline being more utilized than others, the instruction count and invoke parameters do not constitute enough utilization to hide the latency. The reported **average execution cycles is 3509 cycles**, whereas the **calculated WCET is 4000 cycles** based on the analysis in Table 6.3 .

	Instruction	Pipeline	Multiplier	delay	total cycles
1	IMAD	const	1	290	290
		fma	W	4	96
2	S2R	LSU	W	10	240
3	ULDC	LSU	1	290	290
4	SHF	ALU	W	4	96
5	IADD3	const	1	290	290
		ALU	W	2	48
6	ISETP	ALU	W	4	96
7	BRA	CBU	0	0	0
8	S2R	LSU	W	10	240
9	IMAD	FMA	W	4	96
10	IMAD	const	1	290	290
		FMA	W	4	96
11	IMAD	FMA	W	4	96
12	LDG	LSU	1	290	290
13	IMAD	const	1	290	290
		FMA	W	4	96
14	IMAD	FMA	W	4	96
15	FMUL	const	1	290	290
		FMA	W	4	96
16	FMUL	const	1	cached	0
		FMA	W	4	96
17	FMUL	const	1	cached	0
		FMA	W	4	96
18	FMUL	const	1	cached	0
		FMA	W	4	96
19	STG	LSU	1	290	290
					4000

Table 6.3: WCET Analysis of Vectorized Element-wise kernel.

6.3.2.2 Matrix Multiplication Kernel

The matrix dot product kernel exhibits multiple facets of execution paradigms, truly utilizing some of the latency-hiding techniques and memory optimizations for higher performance. Similar to the Vectorized Element-wise kernel, no source code of the implementation is available, but that is not needed for the analysis.

The kernel is relatively big, featuring 99 instructions, counted in Table 6.4. The Kernel was invoked with $8 \times 1 \times 2$ blocks and 128 threads each, which in total results in W=64 on one SM, however, due to the three-dimensional nature of the blocks, 8 blocks will execute on one SM each with 128 threads while one other SM executes the rest resulting in **W=32**. This could be due to the need of creating certain split in the data or creating a certain addressing method.

pipeline	**count**
TC	1
CBU	9
fma	25
alu	31
LSU	
special	3
const	3
global	3
shared	23

Table 6.4: Matrix Multiplication instruction count. LSU instructions are direct memory load or store instructions that are not embedded in other instructions.

Table 6.4 shows some balance of pipelines, mainly fma and alu pipelines, even thoguh fma pipeline is split into fmalite and fmaheavy, but Nvidia reports both pipelines as one, fma, and typically shows a balanced kernel if the fma pipeline has the same amount of instruction as another pipeline. Memory operations are not counted as a pipeline since they are not performed by the cores in an SM. For this purpose, only alu instruction will be counted in this analysis since it has more operations queued. The analysis of alu pipeline is shown in Table 6.5 and the full kernel with the delays is provided in the Appendix. Table 6.5 reflects some important memory access patterns that are described in Algorithm 2, where one constant is loaded for the first time in one instruction, marked as "new const" in the comments' column, and then is reloaded again in another instruction, marked as "reused const." reused constants are as fast as register access, thus marked with 0 additional delay. The reported **average execution cycles is 4083 cycles**, whereas the **calculated WCET is 4774 cycles**.

	Instruction	Pipeline	Multiplier	delay	total cycles	comments
5	ISETP	const	0	290	0	reused const
		alu	32	4	128	
7	ISETP	const	0	290	0	consec. Const
		alu	32	4	128	
9	ISETP	alu	32	4	128	
11	ISETP	const	1	290	290	new const
		alu	32	4	128	
16	IADD3	alu	32	2	64	
20	ISETP	alu	32	4	128	
22	SHF	alu	32	4	128	
23	ISETP	alu	32	4	128	
		const	0	290	0	reused const
25	MOV	alu	32	1	32	
26	ISETP	const	0	290	0	consec. Const
		alu	32	4	128	
27	ISETP	const	0	290	0	consec. Const
		alu	32	4	128	
30	IADD3	alu	32	2	64	
31	IADD3	alu	32	2	64	
33	IADD3	alu	32	2	64	
34	LEA	const	1	290	290	new const
		alu	32	6	192	
35	LEA	const	0	290	0	consec. Const
		alu	32	6	192	
36	LEA	const	0	290	0	consec. Const
		alu	32	6	192	
37	LEA	const	0	290	0	consec. Const
		alu	32	6	192	
40	MOV	alu	32	1	32	
44	ISETP	const	0	290	0	reused const
		alu	32	4	128	
45	ISETP	const	0	290	0	reused const
		alu	32	4	128	
48	ISETP	alu	32	4	128	
51	ISETP	alu	32	4	128	
58	ISETP	alu	32	4	128	
64	ISETP	alu	32	4	128	
70	ISETP	alu	32	4	128	
76	ISETP	alu	32	4	128	
82	ISETP	alu	32	4	128	
88	ISETP	alu	32	4	128	
96	LEA	const	1	290	290	new const
		alu	32	6	192	
97	LEA	const	0	290	0	consec. Const
		alu	32	6	192	
					4774	

Table 6.5: WCET Analysis of Matrix Multiplication kernel. Selected sections of only ALU instruction. consec. = consecutive, within 128 bytes of a previous address. Full analysis is provided in the Appendix.

6.3.2.3 Convolution Kernel

The convolution kernel is last kernel to be processed and is the largest kernel so far, featuring 886 SASS instructions shown in Table 6.6.

Pipeline	Instruction Count	Unique embedded const. loads
fma	400	22
alu	282	9
cba	40	0
lsu		-
special	17	-
const	1	-
global	31	-
shared	40	-
other	75	-

Table 6.6: Convolution instruction count. lsu instructions are direct memory load or store instructions that are not embedded in other instructions. Full analysis is provided in the Appendix.

The fma pipeline is noticeably more utilized, compared to other pipelines, with a total of 400 instructions, 22 of which loads a from a unique constant memory address, and 106 of which loads from a previously loaded constant memory addresses. This, in addition to the low global memory access, and use of shared memory instead of streaming directly from global memory, implies good data reuse patterns in the convolution kernel.

The convolution kernel utilizes 64 registers per thread, this impacts the maximum threads per SM from 64 warps to 32 warps per SM (or 1024 threads per SM), The kernel was called with 29 blocks, each with $8 \times 8 \times 1$ threads, forcing two SMs to each have two warps, as each SM will have $\frac{29 \text{ blocks} \times 64 \text{ thread}}{1024 \text{ thread}} = 1.8$ warp, however, warps can be only whole numbers, thus each SM will have 2 warps, effectively making W=2.

The WCET analysis is done based on the fma pipeline instructions and the fma-embedded constant memory instructions. Due to the large size of the kernel in general, and the fma pipeline instruction count in particular, the analysis was carried out using Excel with the help of formulas, however it is still too large to include in this document. To summarize the used instructions for analysis, 3 fma instructions were used, 192x IMAD instruction, 192x ffma instructions, and 16 fmul instructions, 22 of these instructions load from unique constant addresses than contribute to the delay.

The reported **average execution cycles is 8223 cycles**, whereas the **calculated WCET is 9290 cycles**.

In the following section, the results of the experiments will be further discussed highlighting some execution effects, pitfalls, and limitations.

6.4 Discussion

This work has so far identified the execution characteristics of AI accelerators using systolic arrays and Nvidia GPUs. Systolic accelerators have been expressed in MAC operations rather than cycles due to the lack of information provided by manufacturers in general, and thus only focuses on the general execution style of the systolic arrays shown in Equation 5.6, however, it does not take into account the memory loading time to each PE in the systolic array and does not consider how many cycles for memory loading and for execution of MAC operations.

The experimentations carried out in Section 6.3 for the A100 GPU, and summarized in Table 6.7, are meant to represent some cases of typical kernels used in an AI model, other than the example kernel provided in Section 6.3.1, the used kernels are directly produced by the AI inference library without any compilation or programming style bias. This is important in order to prove no bias in the WCET algorithm in Algorithm 1 and the memory access delay algorithm in Algorithm 2.

Kernel	Average execution time	WCET	Ratio	Analysis table
Example	5145	5298	0.97	6.2
Vectorized Element-wise kernel	3509	4000	0.87	6.3
Matrix Multiplication kernel	4083	4774	0.85	6.5
Convolution kernel	8223	9290	0.88	Appendix

Table 6.7: WCET vs Average execution time.

The Mean Absolute Error (MAE) of the proposed algorithms, expressed as the difference between the estimated WCET and average execution time divided by number of experiments is calculated to be 600.5

$$MAE = \frac{1}{4}\sum_{i=1}^{4} |WCET - \text{Average execution time }| = 600.5$$

represented as a percentage:

$$MAE\% = \left(\frac{600.5}{\frac{5298+4000+4774+9290}{4}}\right) \times 100 = 10.28\%$$

The ratio between the average execution time and WCET comes close to 1 when the kernel is relatively simple, the first example in Table 6.7 shows a streaming kernel with only one pipeline used and no caching behavior whatsoever implying that the minor discrepancy between the average execution time and WCET is due to run-time artifacts. Each SM on the A100 can be guaranteed to be executing the same instruction or experiencing the same memory delay at each step of the execution. On the other hand, the Vectorized Element-wise kernel exhibits a mix of different pipelines and memory operations, but does not saturate any specific pipeline and thus does not hide the latency of other used pipelines.

Matrix multiplication, in general, exhibits high memory access pattern, which would cause

high number of memory delays during execution, however, the matrix multiplication kernel masks the delays by copying the respective values for each kernel from the global memory to both the shared and constant memory. Additionally, the kernel utilizes the alu pipeline more than any other pipeline, effectively masking the delays of other pipelines and memory operations.

Similarly, the convolution kernel copies data from the global memory to the shared and constant memory to optimize access time relies heavily on the fma pipeline. The convolution kernel exhibits multiple instances of the same instruction consecutively, this could be a result of a loop that has been enumerated by the compiler since the filter, input size, and number of channels is constant for that one layer.

The work in [11] proposes a purely dynamic method and a hybrid method, the latter being less accurate than the former as over estimation is noticed in both approaches where the dynamic approach has an average of 102% overestimation while the hybrid approach has an average of 796% overestimation. Whereas the static analysis work done in [21] is claimed to have 26.86% mean absolute error with cases where actual execution times were higher than the predictions.

6.4.1 Limitations

For this work, multiple pieces of data either found by contemporary research papers or by information provided by Nvidia were used to infer execution characteristics of the A100 GPU. However, as far as this research goes, no accurate execution model was described, at least for the Ampere architecture. This work manages to relate the found information, alongside experimentation, to describe a typical execution behavior. Nonetheless, the lack of certainty introduces some limitations that can be summarized as the following:

- Analysis of Systolic accelerators relies on MAC operations rather than cycles, during the development, it was assumed that two types of cycles exist: load and execute. However, the number of cycles doesn't affect the number of MAC operations needed, and can be adapted to fit different systolic array implementations. Whereas the shape of the array and input matrices do indeed affect the number of MAC operations needed.
- Memory delays are not considered for systolic array execution due to the lack of information, but it is believed to be part of the load cycles.
- Nvidia employs multiple levels of latency hiding on the instruction and thread level, however, the more convoluted a kernel is, the harder it is to decode finer implementation of latency hiding. This can be seen in more complex implementations of kernels such as matrix multiplication and convolution.
- Additional research and experimentation is needed to quantify the pipeline multiplier to determine latency hiding of a saturated pipeline in an Nvidia GPU executing a CUDA kernel.
- Kernels used in AI inference applications tend to fit a template of static execution, where loops and iterations tend to depend on the input size and not based on a

variable, and branch divergence could be more impactful in different kinds of kernels. Additional research is needed to estimate the paths and iterations in one kernel

- The algorithm relies on the delays of instructions, which is not publicly available or published by Nvidia and thus one can never have the most accurate algorithm without intensive individual testing of instructions, additionally, the assignment of instructions to pipelines is described to some extent by Nvidia, but not for all instructions. Thus, some assumptions would need to be made regarding some obscure instructions based on the similarities to other instructions and type of data they process.
- Nvidia describes memory and cache behavior to some extent but, to the best of our knowledge, no concrete information or statistical models for cache or memory behavior were found during the creation of this work.
- This work did not cover the execution of multiple concurrent AI models (kernels from different models executing on the same GPU) where the scheduling rules described in Section 4.3.1 would need to be applied. Further research and experimentation need to be applied in this context.

7 Outlook and Summary

The Applications of AI are expanding, becoming more diverse and efficient, and are slowly being normalized in the daily life. Being an integral application of mathematics and computer science, Analysis techniques of computer based systems need to adapt to include AI as part of the analysis. The Worst-Case execution time (WCET) analysis remains one of the most important metrics for many industries, including aviation, automotive, medical equipments, etc. and the need to be able to apply WCET analysis to AI models is becoming more critical than ever.

The approach of this work is to understand the underlying hardware used in the inference operations, However complicated or simple an AI model is, it is executed on real hardware in discrete steps using an instruction set architecture.

Systolic arrays employ a set of a rather simple processing elements that perform multiplication and accumulation, relaying heavily on the physical set-up of these processing elements to facilitate a certain data-flow that produces the result of a matrix multiplication operation. This work finalizes the relationship between the number of these processing elements and how many elements in the matrices to be multiplied to provide a metric of multiply and accumulate (MAC) count, similar to the industry standard of Operations per second (OPS) but independent of time. Memory delay effects and nuances of implementation in systolic arrays were not taken into account, leaving a gap in the Mac_{total} metric that needs to be explored in the future, additionally, concretely correlating the Mac_{total} metric with the cycles of execution that then can be directly embedded in a system-wide heterogeneous WCET analysis.

For Nvidia GPU-based acceleration, an algorithm that takes into account thread-level parallelism, instruction-level parallelism, latency hiding, and memory delays is developed to estimate the WCET of one kernel. Previous works investigate certain aspects of the Nvidia execution paradigm, such as memory access or warp scheduling, and rely on that one aspect as a timing metric for the whole execution cycle of the kernel, which leads to inaccurate estimations for specific scenarios. This work on the other hand correlates multiple execution aspects and execution paradigms and directly deals with the actual SASS instruction set rather than the emulated instruction set (PTX) provided by Nvidia.

The WCET algorithms are designed to process Nvidia Ampere architecture kernels, however, with some modifications to instruction delays and pipelines, the algorithm can be adapted to fit any Nvidia architecture. The instruction delays and associated pipelines need more investigation in the future to provide an accurate set which would provide a tighter WCET estimates and enable more in-depth analysis of additional latency-hiding techniques employed by Nvidia.

Bibliography

[1] A. M. Turing, "I.—Computing Machinery and Intelligence," *Mind*, vol. LIX, no. 236, pp. 433–460, oct 1950. [Online]. Available: https://academic.oup.com/mind/article/LIX/236/433/986238

[2] Y. K. Dwivedi, A. Sharma, N. P. Rana, M. Giannakis, P. Goel, and V. Dutot, "Evolution of artificial intelligence research in Technological Forecasting and Social Change: Research topics, trends, and future directions," *Technological Forecasting and Social Change*, vol. 192, no. April, p. 122579, 2023. [Online]. Available: https://doi.org/10.1016/j.techfore.2023.122579

[3] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "AI Accelerator Survey and Trends," *2021 IEEE High Performance Extreme Computing Conference, HPEC 2021*, pp. 1–9, 2021.

[4] L. Sekanina, "Neural Architecture Search and Hardware Accelerator Co-Search: A Survey," *IEEE Access*, vol. 9, pp. 151 337–151 362, 2021.

[5] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A Survey of Accelerator Architectures for Deep Neural Networks," *Engineering*, vol. 6, no. 3, pp. 264–274, 2020. [Online]. Available: https://doi.org/10.1016/j.eng.2020.01.007

[6] A. Horga, S. Chattopadhyay, P. Eles, and Z. Peng, "Measurement based execution time analysis of GPGPU Programs via SE+GA," *Proceedings - 21st Euromicro Conference on Digital System Design, DSD 2018*, pp. 30–37, 2018.

[7] F. Khorshahiyan, S. K. Shekofteh, and H. Noori, "Predicting execution time of CUDA kernels with unified memory capability," *2019 9th International Conference on Computer and Knowledge Engineering, ICCKE 2019*, pp. 437–443, 2019.

[8] Y. Huangfu and W. Zhang, "Warp-based load/store reordering to improve GPU data cache time predictability and performance," *Proceedings - 2016 IEEE 19th International Symposium on Real-Time Distributed Computing, ISORC 2016*, pp. 166–173, 2016.

[9] Q. Liang, P. Shenoy, and D. Irwin, "AI on the Edge: Characterizing AI-based IoT Applications Using Specialized Edge Architectures," *Proceedings - 2020 IEEE International Symposium on Workload Characterization, IISWC 2020*, pp. 145–156, 2020.

[10] D. Defour, "Measuring Predictability of Nvidia's GPU Schedulers: Application to the Summation Problem," *Proceedings - IEEE 9th International Symposium on Embedded Multicore/Manycore SoCs, MCSoC 2015*, pp. 17–24, 2015.

[11] A. Betts and A. Donaldson, "Estimating the WCET of GPU-accelerated applications using hybrid analysis," *Proceedings - Euromicro Conference on Real-Time Systems*, pp. 193–202, 2013.

[12] H. Kopetz and W. Steiner, *Real-Time Systems*, 3rd ed. Cham: Springer International Publishing, 2022. [Online]. Available: https://link.springer.com/10.1007/978-3-031-11992-7

[13] R. Kirner and P. Puschner, "Classification of WCET analysis techniques," *Proceedings - Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2005*, vol. 2005, no. June 2005, pp. 190–199, 2005.

[14] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem-overview of methods and survey of tools," *Transactions on Embedded Computing Systems*, vol. 7, no. 3, 2008.

[15] R. Amadini, G. Gange, P. Schachte, H. Søndergaard, and P. J. Stuckey, "Abstract interpretation, symbolic execution and constraints," *OpenAccess Series in Informatics*, vol. 86, no. 7, pp. 1–7, 2020.

[16] S. Russell and P. Norvig, *Artificial Intelligence A Modern Approach*, 3rd ed. Pearson Education, 2010.

[17] A. Shrestha and A. Mahmood, "Review of deep learning algorithms and architectures," *IEEE Access*, vol. 7, pp. 53 040–53 065, 2019.

[18] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 2017-Decem, no. Nips, 2017, pp. 5999–6009.

[19] Nvidia Corporation, "CUDA PTX ISA 8.5," 2024. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/ (accessed: March 3, 2025).

[20] Y. Huangfu and W. Zhang, "WCET analysis of the shared data cache in integrated CPU-GPU architectures," *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017*, 2017.

[21] G. Alavani, K. Varma, and S. Sarkar, "Predicting execution time of CUDA kernel using static analysis," *Proceedings - 16th IEEE International Symposium on Parallel and Distributed Processing with Applications, 17th IEEE International Conference on Ubiquitous Computing and Communications, 8th IEEE International Conference on Big Data and Cloud Computing, 11th IEEE International Conference on Social Computing and Networking and 8th IEEE International Conference on Sustainable Computing and Communications, ISPA/IUCC/BDCloud/SocialCom/SustainCom 2018*, no. 1, pp. 948–955, 2018.

[22] T. Raja, "Systolic Array Data Flows for Efficient Matrix Multiplication in Deep Neural Networks," 2024. [Online]. Available: http://arxiv.org/abs/2410.22595

[23] H. Snopce and A. Aliu, "Latency Analysis in the 2-Dimensional Systolic Arrays for Matrix Multiplication," *International Journal of Computers*, vol. 15, pp. 1–7, 2021.

[24] P. Dhilleswararao, S. Boppu, M. S. Manikandan, and L. R. Cenkeramaddi, "Efficient Hardware Architectures for Accelerating Deep Neural Networks: Survey," *IEEE Access*, vol. 10, pp. 131 788–131 828, 2022.

[25] Y. H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," *Proceedings - 2016 43rd International Symposium on Computer Architecture, ISCA 2016*, no. June, pp. 367–379, 2016.

[26] P. Clarke, "Nvidia dominates AI processor market," 2024, [Online]. Available: https://www.eenewseurope.com/en/nvidia-dominates-ai-processor-market-set-to-hit-us100bn-in-2026/ (accessed: March 27, 2025).

[27] Nvidia Corporation, "Nvidia A100 Tensor Core GPU Architecture," 2020. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf (accessed:05.03.2025), pp. 20–21.

[28] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. Donelson Smith, "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed," *Proceedings - Real-Time Systems Symposium*, vol. 2018-Janua, pp. 104–115, 2017.

[29] Nvidia Corporation, "CUDA C Programming Guide," [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (accessed:March 3, 2025).

[30] Keras, "Keras," [Online]. Available: https://keras.io/about/ (accessed: Feb 10, 2025).

[31] Pytorch, "Pytorch," [Online]. Available: https://pytorch.org (accessed: Feb 10, 2025).

[32] Apple, "CoreML," [Online]. Available: https://developer.apple.com/machine-learning/core-ml/ (accessed: Feb 10, 2025).

[33] tensorflow, "TensorFlow," [Online]. Available: https://www.tensorflow.org (accessed: Feb 10, 2025).

[34] A. J. Abdelmaksoud, S. Agwa, and T. Prodromakis, "DiP: A Scalable, Energy-Efficient Systolic Array for Matrix Multiplication Acceleration," pp. 1–11, 2024. [Online]. Available: http://arxiv.org/abs/2412.09709

[35] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and Evaluation of the First Tensor Processing Unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, 2018.

[36] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim," *Proceedings - 2020 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020*, pp. 58–68, 2020.

[37] Nvidia Corporation, "Nvidia Turing GPU," [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (accessed: 10 Jan, 2025).

[38] ——, "Nvidia Nsight Compute," [Online]. Available: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html (accessed: 03 February 2025).

[39] ——, "Nvidia Tesla V100 GPU Volta Architecture," [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (accessed: 16 Jan, 2025).

[40] Y. Arafa, A. H. A. Badawy, G. Chennupati, N. Santhi, and S. Eidenbenz, "Low overhead instruction latency characterization for NVIDIA GPGPUs," *2019 IEEE High Performance Extreme Computing Conference, HPEC 2019*, 2019.

[41] H. Abdelkhalik, Y. Arafa, N. Santhi, and A. H. A. Badawy, "Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis," *2022 IEEE High Performance Extreme Computing Conference, HPEC 2022*, 2022.

[42] Nvidia Corporation, "CUDA Binary Utilities," [Online]. Available: https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#ampere-ampere-instruction-set-table (accessed: 03 February 2025).

[43] H. Hamm-Lippstadt, "5G Campus Networks as an Enabler for Real-Time Learning in Organic Farming," [Online]. Available: https://www.hshl.de/forschung-unternehmen/forschungsprojekte/forschungsprojekte-im-themenfeld-eingebettete-systeme/5g-landwirtschaft-ml/ (accessed: 11 March 2025).

[44] A. Alhalabi, "Appendix: Characterization of Artificial Intelligence Accelerators for Timing Analysis," [Online]. Available: https://github.com/obi-two-kenobi/Characterization-of-Artificial-Intelligence-Accelerators-for-Timing-Analysis/ (accessed: 12 May 2025).

List of Figures

List of Tables

Affidavit

Ich versichere, dass ich meine Arbeit selbstständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.
Dortmund, May 12, 2025

Ali A. K. Alhalabi

Affidavit

I hereby confirm, that I have written the Master Thesis at hand independently, that I have not used any sources or materials other than those stated, and that I have highlighted any citations properly.
Dortmund, May 12, 2025

Ali A. K. Alhalabi

A Appendix

The Appendix of this work is hosted on GitHub [44], the hosted documents are in the following structure:

1. **Analysis.xlsx**: The excel file that contains the full analysis of each kernel. Each kernel analysis is carried out in a separate sheet and are arranged as the following:
    - **Introductory example page**: Contains the analysis of the introductory example. Metrics of this kernel are collected in *Add kernel execution metrics.ncu-rep* and based on which the results were compared.
    - **Vectorized element-wise**: Contains the analysis of the first Kernel of the AI model, with ID 0 in *AI model execution metrics.ncu-rep*.
    - **Matrix multiplication**: Contains the analysis of a matrix multiplication Kernel of the AI model, with ID 173 in *AI model execution metrics.ncu-rep*.
    - **Convolution**: Contains the analysis of a convolution Kernel of the AI model, with ID 17 in *AI model execution metrics.ncu-rep*.
2. **CUDA Occupancy Calculator.xls**: The occupancy calculator provided by Nvidia as a separate Excel sheet. Also available as part of Nvidia Nsight Compute.
3. **runAIModel.py**: Python script used to execute the used AI model, the script was used as command inside Nvidia Nsight Compute.
4. **Nsight Compute Reports**:
    - **AI model execution metrics.ncu-rep**: AI model kernels report produced by Nvidia Nsight compute.
    - **Add kernel execution metrics.ncu-rep**: Introductory Example report produced by Nvidia Nsight compute.