TÉCNICO LISBOA

Exploiting GPU Undervoltage to Improve the Energy Efficiency of Deep Learning Applications

Rafael Ferreira Gil

Thesis to obtain the Master of Science Degree in

Eletrical and Computer Engineering

Supervisors: Doutor Nuno Filipe Valentim Roma
Doutor Pedro Filipe Zeferino Tomás

Examination Committee

Chairperson: Doutora Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Doutor Pedro Filipe Zeferino Tomás
Member of the Committee: Doutor Nuno Cavaco Gomes Horta

January 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First of all, I would like to thank my parents for their encouragement, caring, and sacrifices over all these years. To them, I owe what I am today and all I have achieved. Furthermore, I would like to thank my brother Gonçalo for his availability, patience, and friendship. Also, a special thank you to Mariana whose support and encouragement were crucial to me.

Additionally, I would also like to thank my dissertation supervisors Professor Nuno Filipe Valentim Roma and Professor Pedro Filipe Zeferino Tomás for their insight, support, and sharing of knowledge that has made this Thesis possible.

Finally, I want to thank my friends, colleagues, and all whose path has crossed mine and contributed to my personal and professional growth.

Abstract

The success of deep learning applications, within machine learning and artificial intelligence, is pushing further this area's development. However, the increasing performance and accuracy needs are usually met with higher computational requirements, whose efficiency is, more often than not, disregarded. General Purpose Graphics Processing Units (GPGPUs), being the state-of-the-art accelerators for these applications, play a significant role in making deep learning models widely available. However, the large power consumption increases operational costs and eschews resource-constrained environments from using such devices. To mitigate this problem, the present work proposes an approach to study the potential energy savings of reducing the supply voltage of those devices, using an AMD Radeon Vega Frontier Edition GPGPU. This endeavor is first applied to synthetic benchmarks to characterize the device's voltage guardband and then to current deep learning models to provide an insight into their behavior under minimum supply voltage. Results show deep learning models can achieve energy savings of up to $24.79\,\%$ (average of $15.35\,\%$) and still guarantee their initial accuracy. Nonetheless, the energy savings can be further increased up to $30.16\,\%$ (average of $18.37\,\%$) at the expense of the model's accuracy. Deep learning applications experienced an accuracy droop up to $61.52\,\%$ (average of $10.61\,\%$) when working at near failure supply voltage.

Keywords

Voltage guard band, Graphics Processing Unit, Energy efficiency, Deep learning

Resumo

A crescente procura por aplicações de aprendizagem automatiza está a estimular o aumento da performance das mesmas. No entanto, este incremento de performance é normalmente alcançado à custa de um aumento dos requisitos computacionais, cuja eficiência energética é, por norma, ignorada. Dispositivos de Processamento Gráfico de Propósito Geral (GPGPU), sendo os aceleradores de eleição para este género de aplicações, têm um papel importante na sua disponibilização global. Todavia, estes dispositivos apresentam consumos energéticos elevados que para além dos custos operacionais que implicam, colocam ambientes com recursos limitados à margem destas tecnologias. Para mitigar este problema, no âmbito desta tese, é proposta uma abordagem para estudar potenciais ganhos de energia obtidos ao reduzir a tensão de alimentação de GPGPUs, usando o dispositivo *AMD Radeon Vega Frontier Edition*. Este processo é inicialmente reproduzido utilizando aplicações padrão com o objetivo de caracterizar a margem de tensão do dispositivo imposta pelo fabricante. Depois, é também reproduzido utilizando modelos atuais de aprendizagem automática permitindo conhecer o seu comportamento sob baixas tensões de alimentação. Os resultados mostram que os modelos de aprendizagem automática podem atingir eficiências energéticas que chegam aos $24.79\,\%$ (média de $15.35\,\%$) e ainda assim garantir a precisão inicial. Não obstante, é ainda possível obter melhores eficiências energéticas, até um máximo de $30.16\,\%$ (média de $18.37\,\%$) sob pena de perda de precisão do modelo quando o dispositivo GPGPU trabalha a níveis de tensão próximos do seu colapso.

Palavras Chave

Margem de segurança da tensão de alimentação, Unidade de Processamento Gráfico, Eficiência energética, Aprendizagem automática

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

AMD	Advanced Micro Devices, Inc.
AGT	AMD GPU Tool
API	Application Programming Interface
ALU	Arithmetic Logic Unit
AI	Artificial intelligence
BLAS	Basic Linear Algebra Sub-routine
BERT	Bidirectional Encoder Representations from Transformers
CPU	Central Processing Unit
CU	Compute Unit
CUDA	Compute Unified Device Architecture
CNN	Convolution Neural Network
DRAM	Dynamic Random-Access Memory
DVFS	Dynamic Voltage and Frequency Scaling
FET	Field Effect Transistor
FP	Floating Point
FCN	Fully Connected Neural network
GCN	Graphics Cores Next
GRU	Gated Recurrent Unit
GPGPU	General Purpose Graphics Processing Unit

GPC	Graphic Processing Cluster
HCC	Heterogeneous Compute Compiler
HIP	Heterogeneous Interface for Portability
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IPC	Instructions Per Clock
IBM	International Business Machines Corporation
LSTM	Long short-term memory
LPP	Low Power Plus
MOS	Metal Oxide Semiconductor
MRPC	Microsoft Research Paraphrase Corpus
NLP	Natural Language Processing
NBTI	Negative Bias Temperature Instability
NMOS	n-type Metal Oxide Semiconductor
OpenCL	Open Computing Language
ONNX	Open Neural Network Exchange
PBTI	Positive Bias Temperature Instability
PV	Process Variation
PMOS	p-type Metal Oxide Semiconductor
RAW	Read-after-Write
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SDC	Silent Data Corruption
SIMD	Single Instruction, Multiple Data
SPU	Special Processing Unit
SM	Streaming Multiprocessor

TDP Thermal Design Power

WAR Write-after-Read

WAW Write-after-Write

ROCm Radeon Open Ecosystem

1

Introduction

Contents

Modern applications leveraged by deep learning technologies are feasible due to hardware accelerators. In particular, General Purpose Graphics Processing Unit (GPGPU) devices are widely used in this context given their unique characteristics. They are easier to configure taking advantage of high-level programming languages, there is also an extensive user base community behind these devices, and additionally, they are largely available in data centers and cloud-based services. Consequently, GPGPU devices are considered the main accelerator for deep learning applications [1].

However, there is a high demand for performance improvements in deep learning applications. But, more often than not, the performance enhancement is met with increasingly higher computational requirements. As a result, there is a trade-off between deep learning application's performance and the available computing capabilities of GPGPU accelerators.

The literature on deep learning improvement techniques for GPGPU devices is also rapidly growing. However, the majority of these works neglect the power consumption impact of the proposed solutions. The increasing power consumption is a major challenge on the GPGPU device's usage, especially when considering resource-constrained environments. As a result, there is a need for performance improvements to be weighed against their energy overhead, and for a deeper dive on lower precision techniques that impose a trade-off between accuracy and efficiency [2].

GPGPU manufacturers design their processing units bearing in mind a given performance standard. Naturally, each architecture will have its own minimum required supply voltage for the device to work properly. However, taking into account phenomena that can negatively impact the device, manufacturers add an extra safety margin on top. This opens up a window to improve the GPGPU devices energy efficiency, given that the power consumption is deeply related to the supply voltage [3, 4].

1.1 Objectives and Contributions

This thesis proposes an approach to study the potential energy savings of GPGPU devices on modern deep learning applications by reducing the voltage guardband margin imposed by the manufacturers.

However, due to the limitations of NVIDIA devices for the prosecution of this work, it will focus on the Advanced Micro Devices, Inc. (AMD) Radeon Vega Frontier Edition GPGPU device.

The main goal of this thesis is to achieve energy efficiency in deep learning applications. To accomplish this, the following objectives are proposed:

- Characterization of AMD Radeon Vega Frontier Edition GPGPU card voltage guardband. Motivated by the potential energy saving results and the lack of literature featuring AMD devices.
- In-depth performance and energy efficiency evaluation regarding the proposed approach .

- Evaluation of the accuracy impact of the voltage guardband reduction approach on modern deep learning applications.

1.2 Outline

The remaining of this thesis is organized into four chapters. Chapter 2 introduces deep learning applications and provides an overview of existing solutions to attain energy efficiency. Chapter 3 contains a characterization of the GPGPU voltage guardband through two sets of benchmarks. Chapter 4 applies the knowledge from chapter 3 into deep learning applications providing an insight into the voltage guardband impacts on their execution. Finally, chapter 5 provides the conclusions of the current thesis along with future work opportunities.

2

Background and related work

Contents

2.1 Deep Learning

Artificial intelligence (AI) is the capability of machines to exhibit human-like intelligence. However, a machine with all our reasoning and consciousness is not the only definition that fits. This is actually, a very specific branch of AI called *General AI*, which up until now, remains fiction. Nonetheless, *Narrow AI*, another branch of AI, is currently feasible given that there are machines that can perform a specific task with even greater ease and efficiency than a human. From computer vision and speech recognition to self-driving cars, all are good examples of this technology.

This kind of machine can be developed using several different approaches. One of them is a hard-coded approach, where the programmer has to define all the decisions the machine has to take within all the different scenarios it might encounter. This is hard to accomplish, as the definition of all the different scenarios might be impossible to consider since they can be unpredictable.

The ambition of creating something that can think dates back to ancient Greece [5]. The real problem is the definition of learning and understanding the human learning capability so that it could be replicated by a computer. This is itself, a scientific goal [6]. Anyway, all the effort in this area has led to a new approach to AI called machine learning, which already existed around the decade of 1950 but was only theoretical or required special experimental hardware [5].

The learning effect is achieved through the definition of abstract concepts that are created in this parsing phase. However, this still requires a lot of hand-coded work to be done, to develop detectors, such as edge or shape detectors. Despite being promising, this approach still did not allow the development of outstanding applications, even in the *Narrow AI* branch.

Nonetheless, those first steps on machine learning allowed the development of a new stand in the AI field: Deep Learning. Deep learning, as it derives from machine learning, is also a solution that allows machines to learn, by using the same principle based on the definition of abstract concepts is applied. The key difference is that, in deep learning, a hierarchy of concepts is defined, thus creating a concept sequence that depends on each other. This sequence implies that the furthest in the hierarchy the representation is the more abstract and complex a given concept would represent [5]. This hierarchy can have multiple stages, hence the name deep learning.

2.1.1 Neural Network

To implement deep learning, neural network solutions, which are inspired by the brain structure and its ability to learn, are used. Naturally, these neural networks do not work the same way our brains do, but there is no reason to believe that the way humans learn is the only way to acquire knowledge [6]. A neuron, in this context, is no more than a fraction that represents an activation. Given the hierarchy architecture of deep learning, these neurons are organized in layers and the activation of

a given layer's neuron is determined by the activations, i.e., the neurons, of the previous layer. This behavior is analogous to biological brain cells, which are triggered when some other set of cells has been triggered.

Figure 2.1 shows an illustration of a deep learning neural network model. Here, the input layer is composed of the pixels of the input image, that is the first layer neurons' activation corresponds to the values of each pixel from the input image. In this example, and also the majority of the models, only the first layer is visible. Given the input pixels in the first layer, the second layer can identify other features, such as edges. Given the second layer, the third one can identify corners or contours. This could continue, with the identification of specific parts of an object and even further. In this example, the last layer, which is the output of the model, could be a score between classes that in this case allowed the identification of a cat.

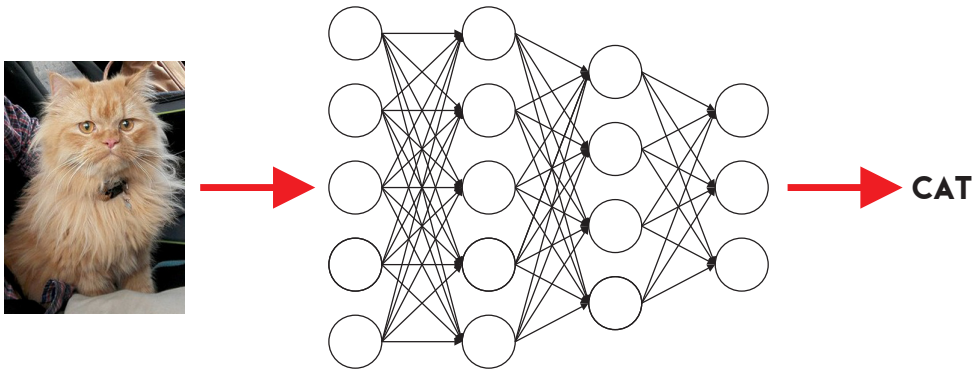


Figure 2.1: Illustration of a deep learning model.

There are two main different families of neural networks: feedforward neural networks and recurrent neural networks. Feedforward neural networks, or multilayer perceptrons, are used to approximate some function f^*, allowing, for instance, a classifier that maps from an input x to some class y, defined by $y = f^*(x)$. This network approximates f^* by defining its own f function, which depends on the input x and also on the parameters θ, $y = f(x;\theta)$ [5]. The name "feedforward" results from the fact that there is no feedback between the output and any intermediary stage of the network, meaning that there are no loops between the network's neurons. On the other hand, when a network's output is fed into itself a recurrent neural network is obtained. The example in figure 2.1 is a feedforward network, as there are no loops in its model.

Given the layered architecture of the neural network, one can conclude that function $f(x,\theta)$, which approximates $f^*(x)$, is given by $f(x,\theta) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(x,\theta)\right)\right)$, where the functions $f^{(1)}$, $f^{(2)}$ and $f^{(3)}$ would correspond to the operations that are made in the the first, second and third layer of the network, respectively.

2.1.1.A Layers

A fully connected layer connects all inputs to all the defined outputs through a weight per connection plus a bias. It can be viewed as matrix multiplication, as exemplified in the following equation 2.1, by considering n inputs and m outputs.

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{2.1}$$

The fully connected layer is usually used in the last layer of a network, because it correlates all the previous layer neurons, that at this time represent some high-level abstract feature, with the desired output. This allows the definition of how much impact each high-level feature has on each specific output.

A convolutional layer is the most used layer in a Convolution Neural Network (CNN), which belongs to the feed-forward neural network family. These networks are currently the best approach towards computer vision, having obtained good results in the field [7, 8]. The way this layer works is almost self-explanatory: it outputs the convolution result between the input and a given kernel. The kernel size of this convolution usually varies from 1×1 up to 7×7, but ultimately is the model designer who chooses its size. Other parameters are up to the model designer to define, such as the stride and the padding of the convolution. It is important to note that the kernel depth must match the depth of the input. For example, if an RGB image is used as input, the kernel must have a depth of 3, i.e. the format $x \times x \times 3$, to cover the 3 channels of an RGB image.

Since the convolutional or the fully connected layers both provide linear operations, it would not be possible to define a non-linear function model using those layers exclusively. Therefore, different activation layers are added to the model's architecture specifically to introduce a non-linearity. One, of such layers, is the Rectified Linear Unit (ReLU) which is widely used in CNNs. Naturally, depending on the model's end goal different activation layers might be chosen.

Nonetheless, the operation on a ReLU layer consists of the application of a function as the one present in Figure 2.2. Even though the ReLU activation function is not globally linear it is the composition of two linear functions, and as result, many of the properties that make linear models easier to optimize are preserved. This property contrasts with activation functions such as the $tanh$ and $sigmoid$, which still introduce the desired non-linearity, but have other problems of their own, e.g. the vanishing gradients [5, 9].

Finally, the pooling layer, which is also denoted by the downsampling layer consists of defining a filter of a given custom size (3×3 for instance) and of a stride of the same length. Then, it goes through the input of the layer and applies a max, an average, or some other pooling function. An example of a

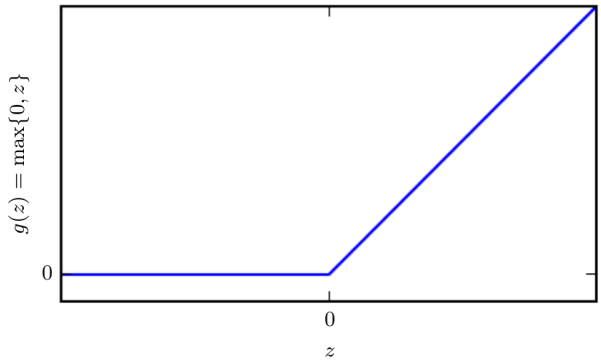


Figure 2.2: ReLU function graph.

max-pooling layer is shown in Figure 2.3. The purpose of this layer is to reduce the computation cost of the following layers, by drastically reducing the number of outputs. In the context of computer vision, this layer is highly viable after some convolutional layers, because after a given feature is identified in the previous layers (i.e., some inputs have high activation values), the exact location of that occurrence is not so important.

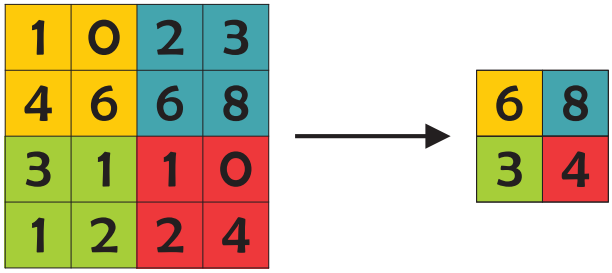


Figure 2.3: Example of a max-pooling layer with 2×2 filter.

Unlike a feedforward neural network, a recurrent neural network can receive as input a sequence of data and produce also a sequence of outputs. This property makes them important for applications such as speech recognition or video classifiers. Long short-term memory (LSTM) is a recurrent neural network layer that can be stacked, similarly to the convolution layer in a CNN. Figure 2.4 depicts such a layer diagram.

This layer has a neuron that acts as a memory cell preserving some past input for as long as ”keep gate” determines. ”Keep gate” is also a neuron that defines how long the memory cell information should be preserved. This layer contains two more neurons called ”read gate” and a ”write gate” which define when read and write access to the memory cell should be granted, respectively. This way, since the memory cell is fed into itself, this layer has the capability of storing information and thus adapting to the sequence at the input.

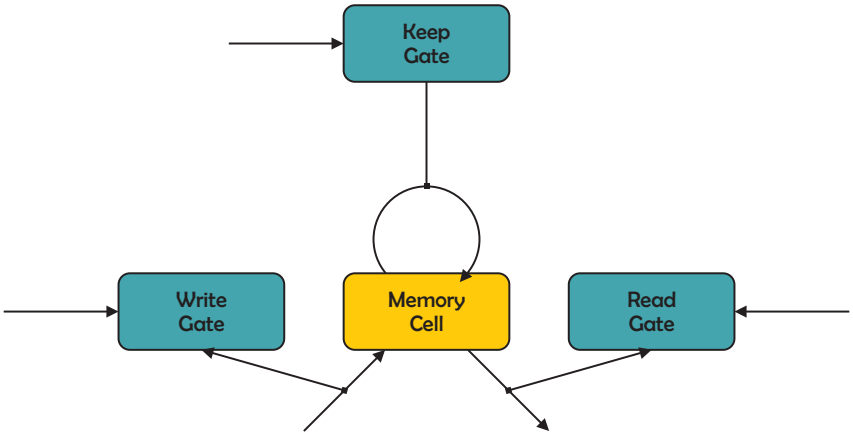


Figure 2.4: LSTM RNN architecture.

2.1.1.B Execution Phases

There are two different execution phases a deep learning model goes through. They are the *training* and the *inference* phase. The training phase is where the model learns what the expected result is based on a set of input data. The inference phase occurs after the training phase is completed and the model is deployed, to infer the output of some real-world input. In supervised learning, it does so by defining the θ parameters that would best approximate the desired function f^*. To achieve this approximation, two important concepts are needed: a cost function and an optimizer function.

A cost function, also called the loss function, is a measure of how far the model is from the intended result, given by $f^*(x)$. The mean squared error and the negative log-likelihood are both examples of this kind of function, the latter being one of the most widely used in this context.

The job of the optimizer is to minimize the previously defined cost function in each iteration of the training phase, by redefining the model's parameters. This is done through a process called backpropagation, since the model's real and desired output for a given input is known, the error can be calculated using the cost function. Afterward, the error is backpropagated into the model, now updating the models' parameters so that the error can be minimized. This process is repeated until the desired accuracy is achieved.

A decrease in the cost function means the accuracy of the model is increasing and thus the model is learning. However, this is not as straightforward as it sounds, due to a phenomenon named *overfitting*. Accordingly, the accuracy of a model should be measured with a set of data, the test data set, that does not belong to the training data set, to check the model's ability to extrapolate from the training data to real data. A model can have a high accuracy rate within the training data set, but perform poorly with data it has not seen before. This is the definition of overfitting. Hence, a decrease in the cost function means the model's accuracy increased solely concerning the training set. To attain this objective, the training phase of a model should be held while both the training data and the test data accuracy are

increasing. Once the test data accuracy stops increasing and the other keeps going up, overfitting is starting to happen.

To prevent the model from overfitting, there is a special layer that is often particularly useful: the dropout layer [10]. This layer is only used while the model is in the training phase and it simply drops out a random set of activations in its input, by setting them to zero. This forces the model to output some result even if activations, considered relevant for that output, were dropped out.

The optimization procedure is what grants a deep learning model the ability to learn. This optimization procedure consists of finding the global minimum of the cost function. Sometimes, this algorithm can get trapped in a local minimum and therefore not converge towards the desired point. That is why the choice of this procedure is important. Depending on the gathered information about the cost function, many different optimization procedures can be used (e.g., function values, first derivatives, or second derivatives).

The most commonly used optimization procedure is gradient descent. In order to minimize a given function $y = f(x)$, this iterative method starts at a random value of x. Then, at each iteration, it moves towards the symmetrical value of the derivative of that function at the current value of x multiplied by a constant named learning rate, η, which defines how fast the method converges. The mathematical expression of this method is defined in the equation 2.2.

$$x^{(t+1)} = x^{(t)} - \eta \cdot \nabla f\left(x^{(t)}\right) \tag{2.2}$$

Naturally, this method does not converge in all cases, and the definition of η can influence its convergence rate. There are several optimizations to this method available that would allow a more robust and faster convergence, such as the momentum technique or the adaptive step sizes.

In short, almost all deep learning algorithms follow the same recipe: a dataset, a cost function, an optimization procedure, and a model. The particular implementation of each of these pieces is what differentiates one approximation from another [5].

2.1.2 Deep Learning Frameworks

Deploying state of the art deep learning models by a researcher or engineer would require a long developing period if no specialized toolboxes were available, given the amount of work developed in the area.

Hence, a deep learning framework provides an environment where deep learning algorithms can be expressed, thus defining and subsequently executing a computation graph. A computation graph is the collection of operations that must be executed in a certain order to obtain a given result. An example of a simple neural network computation graph is depicted in Figure 2.5.

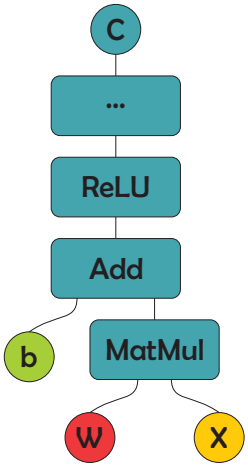


Figure 2.5: Example of the neural network computation graph.

Defining the computational graph, by providing the framework's users tools to do so in an efficient way is the main focus of a deep learning framework. However, the efficient execution of the computational graph is also decisively important. To accomplish that, deep learning frameworks resort to libraries that already implement the needed operations efficiently allowing framework developers to focus on their own paradigm instead of low-level optimizations. Furthermore, such libraries can provide abstractions from specific hardware or operating systems allowing easy portability.

The main libraries used by deep learning frameworks are efficient mathematical implementations required by deep learning algorithms both with and without GPGPU accelerations, depending on the support a given framework provides. Basic Linear Algebra Sub-routines (BLAS) and specific deep learning algorithms libraries are both a must-have in this kind of frameworks, being cuBLAS and cuDNN examples of such libraries.

Given the popularity increase of deep learning models, several specialized frameworks were developed to address this necessity. Currently, there are many frameworks available, and selecting one can itself be a complicated task. The framework stability, reliability, and ability to keep up to date with new developments are all factors to take into account in this process.

The table 2.1 lists some of the most widely adopted frameworks available for deep learning development, which fit the mentioned criteria.

2.1.2.A Torch

Torch is one of the oldest machine learning frameworks, which has been available since 2000. Torch allows setting up, training, and running a deep learning network by configuring its hyper-parameters, enabling the development of a large number of deep learning models. It was initially implemented in C++ by following a modular strategy, permitting further modifications and new algorithm extensions [12].

Table 2.1: Deep learning frameworks [11].

FRAMWORK	CORE LANGUAGE	BINDINGS	CPU	GPU	PRE-TRAINED MODELS
Torch	Lua		•	•	
PyTorch	Python		•	•	
Caffe	C++	Python, Matlab	•	•	•
TensorFlow	C++	Python	•	•	•
CNTK	C++	Python, C#, Java	•	•	•
MXNet	C++, Python	Java, R, Scala, Javascript	•	•	•

However, from Torch3 on, the development was ported to the Lua programming language.

Lua is an embeddable scripting language that focuses on lightweight. This makes it highly suitable for resource-constrained environments, such as embedded systems. Lua was written in C and is known for its fast performance. This language is best known within the gaming community, but not so much by the general public, which offers a big barrier to entry since access to support is reduced and also the number of third-party libraries is limited.

Also, Lua is easily embeddable, which means an effortless integration into a final product or system is granted after the program itself is written. And it also targets a wide range of devices and operating systems. For these reasons, Torch developers chose Lua, as it satisfies their main constraints out of the box: easy extendability and fast performance [12].

2.1.2.B PyTorch

To circumvent the Lua programming language barrier, a group of developers, inspired by Torch's programming style, implemented Torch in python, naming it PyTorch. First of all, PyTorch is not a python wrap around another language, as many frameworks that support python, are. This framework is deeply integrated with python and implements two paradigms that might be the reason behind its popularity rise: imperative programming and consequently dynamic computation graphs.

Imperative programming means computation statements are executed right away, thus immediately modifying the program's state. On the other hand, symbolic programming means explaining what a program should do without explicitly stating how it should be done, leaving that implementation up to a compiler. Thus, creating a separation between defining and executing a computation. Algorithms 2.1 and 2.2 depict the contrast between these programming paradigms. In sum, symbolic programs first define all computations to obtain a given result and only then execute them, that execution is represented in line 6 of the algorithm 2.2, whereas imperative programming explicitly defines each step towards the result. Naturally, symbolic programs are more efficient, since management of the program's control flow and its required resources can be made, whilst imperative programs are more flexible and allow easier debugging and understanding of stack trace reports.

Algorithm 2.1 Imperative programming

1: $x = variable(x)$
2: $y = variable(y)$
3: $z = x + y$ ▷ The value of z is calculated once this line is executed
4: $w = z \times x$ ▷ The value of w is calculated once this line is executed

Algorithm 2.2 Symbolic programming

1: $x = variable(x)$
2: $y = variable(y)$
3: $z = x + y$ ▷ Defines how the variable z is going to be calculated, but its result is not calculated yet
4: $w = z \times x$ ▷ Defines how the variable w is going to be calculated, but its result is not calculated yet
5:
6: COMPILE(w) ▷ Only here, the computational graph is executed, thus obtaining the result of w

The dynamic computation graph means PyTorch generates its computation graph structure at runtime, whereas a static computation graph paradigm means the computation graph is first defined and only then executed, with the advantage of allowing graph optimizations before its execution, analogously to imperative programming. However, for some use cases, such as recurring neural networks, it would be useful if the computation graph could change depending on the input data.

2.1.2.C Caffe

Caffe is also one of the oldest deep learning frameworks and was developed by Berkeley AI Research. It became open source in 2014 and currently supports all major desktop platforms.

This framework is based on a symbolic programming approach since there is a separation between the models' representation and implementation. This framework was originally developed for computer vision tasks, which suit well feedforward neural networks and more especially CNNs. Nonetheless, newer versions expanded their development to speech and text recognition too. It also supports all stages of the deep learning model's development, from training to deployment.

A large set of pre-trained models and also the code to reproduce them is available through Caffe's community, providing, this way, a starting boost since the expensive learning phase is avoided [11]. Caffe is fully implemented in C++ and Compute Unified Device Architecture (CUDA) through the cuDNN library, offering a smooth transaction between Central Processing Unit (CPU) and GPGPU executions. Python and MATLAB interfaces are also provided.

In Caffe's computation graph, each node is considered a layer, and a new layer definition requires its full behavior description, such as the forward and backward gradient updates. This makes Caffe quite inflexible to changes. However, to ease the development, a long list of different layers is provided by the Caffe community, even though this does not solve the problem, only mitigates it.

2.1.2.D TensorFlow

TensorFlow was developed by Google, based on their previous proprietary deep learning project called DistBelief, to allow an efficient implementation of large scale models for a variety of different devices, ranging from resource-constrained environments to systems consisting of hundreds of machines. In essence, Tensorflow would improve deep learning algorithms' portability, to ease their commercial use [13].

TensorFlow's core is developed in C++ and a Python API wrapping around it is provided. This way, heavy computations are executed in a high-performance programming language while models' descriptions can be done in a high-level programming language. Tensorflow provides both a symbolic and imperative programming approach.

In contrast with the Caffe framework, each TensorFlow computation graph's node represents either data or a math operation. Hence, a layer can be defined as a set of nodes. This makes TensorFlow building blocks smaller, which translates to more modularity and less programming verbose. Edges, in this computation graph, represent the flow of data between nodes and this data that flows through nodes is a multi-dimensional array called a tensor, hence the name TensorFlow. The output of one operation is then fed into the next until the result is obtained. Despite TensorFlow being developed to support neural networks, it can be used by any application whose computation could be modeled as a data flow graph.

GPGPU acceleration is supported, currently only for Nvidia devices, through the cuDNN library. Nonetheless, Open Computing Language (OpenCL) support is foreseen in the future, which will allow TensorFlow GPGPU acceleration on non-Nvidia devices. For algebra implementations, TensorFlow uses cuBLAS and Eigen libraries, which are implementations of Basic Linear Algebra Sub-routines (BLAS) with and without GPGPU support respectively.

2.1.2.E MXNet

MXNet is an open-source deep learning framework developed by Apache. One of its main focuses is scalability allowing for a distributed deployment into a cloud infrastructure. Furthermore, MXNet allows developers to use both imperative and symbolic programming, depending on their preferred programming style and offers a wide range of programming languages, including C++, Python, Julia, Matlab, JavaScript, Go, R, Scala, Perl, and Wolfram [14].

2.1.2.F CNTK

Microsoft Cognitive Toolkit, also known as CNTK, is a deep learning framework developed by Microsoft. It supports multiple programming languages, such as Python, C++, C#, and Java.

This framework eases the integration of deep learning applications into Microsoft's products, such

as Azure. It uses a direct computation graph to describe a neural network and can be used to create a wide range of models, including CNNs and RNNs. CNTK also handles the neural network's learning stage automatically as soon as it is modeled.

The latest and final version of CNTK was published in April 2019. It was one of the first frameworks to support Open Neural Network Exchange (ONNX): a format representation of deep learning models. An advantage of this format is allowing interoperability between frameworks, meaning the model can be deployed to different execution engines out of the box [15].

2.1.2.G Other Frameworks

Some frameworks rely on existing frameworks as engines. For example, Keras is a framework, released in 2015, that initially was executed on top of TensorFlow or a few other frameworks depending on the user preferences, depending on the developer's choice. However, after version 2.4, Keras fully integrates with Tensorflow only working as an interface to it.

Keras uses an object-oriented design so all the components of the other frameworks are considered objects. This provides a cleaner and easier to understand interface. As a drawback, the full functionality of Tensorflow is not available through Keras.

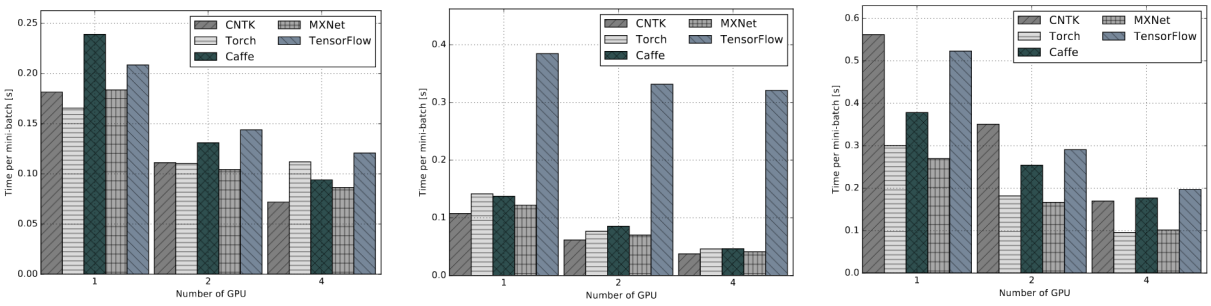
Keras is better optimized for usage with Theano. Theano is also a deep learning framework deeply similar to TensorFlow since its computations could also be modeled as a data flow graph. However, Theano, whose first release dates back to 2007, has been recently discontinued.

2.1.2.H Deep learning frameworks benchmarking

Given the mentioned deep learning frameworks, they have their architectural similarities and differences. However, there are other concerns to take into account, such as the performance, when it comes to selecting one either for research or commercial development. With that in mind, a study was made at the beginning of 2017, featuring current CPU and GPGPU devices, to obtain a running time performance comparison of such frameworks [16].

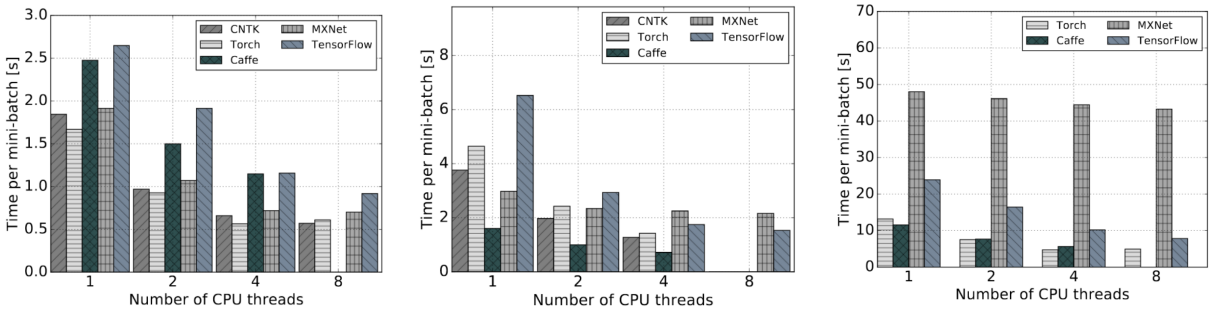
This study covered the execution of three different neural networks: Fully Connected Neural network (FCN), CNN, and RNN. Overall results are depicted in Figures 2.6 and 2.7

The performance scalability is poor when using CPU, except for TensorFlow that scales better than the other tested frameworks. Using only a single GPGPU, Caffe, CNTK, and Torch surpass MXNet and TensorFlow's performance. But in general, when using GPGPU acceleration all frameworks obtained a significantly higher efficiency when compared to the CPU implementation. However, it is important to notice Torch's remarkably scaling capacity when using AlexNet, a CNN [16].



(a) FCN-R **(b)** AlexNet **(c)** ResNet

Figure 2.6: Performance comparison on multi-GPU platform [16].



(a) FCN-R **(b)** AlexNet **(c)** ResNet

Figure 2.7: Performance comparison on Intel i7-3820 CPU [16].

2.2 General Purpose Graphics Processing Unit

GPGPU devices are known to be the state of the art in both the training and inference phase of deep learning neural networks as they provide high levels of speed and energy efficiency [17]. However, the search for neural networks' accuracy empirically leads to increasing computational requirements as the number of parameters and hidden layers grow larger.

The computation requirements of the training phase surpass inference by several orders of magnitude due to a high number of iterations and gradient calculations. Therefore, the training phase cost considering an extremely large neural network, with over 1 billion parameters, becomes overwhelming. Nonetheless, such a large neural network was trained in just a few days using a system built with TensorFlow predecessor, DistBelief, which used 1000 machines and a total of 16 000 CPU cores [18]. However, the vast majority of researchers and developers have no access to that amount of resources. Notwithstanding, another system with just 3 machines and adopting GPGPUs to leverage computing capabilities managed to train a neural network 6.5 times larger, with over 11 billion parameters, within the same amount of time [19].

2.2.1 Parallel computing

A GPGPU is a specialized hardware component designed to allow the exploitation of massive data parallelism. For instance, image rendering is an application that benefits from these devices given its high-level parallelism.

To program these kinds of devices several Application Programming Interface (API), are available, such as CUDA and OpenCL. CUDA and OpenCL both provide low-level access to GPGPUs, which ultimately translates to better performance ratings.

CUDA is a proprietary API developed by Nvidia which is only available for their own devices. On the other hand, OpenCL is open-source and can run on a wider set of devices. Notably, translations between CUDA and OpenCL are straightforward.

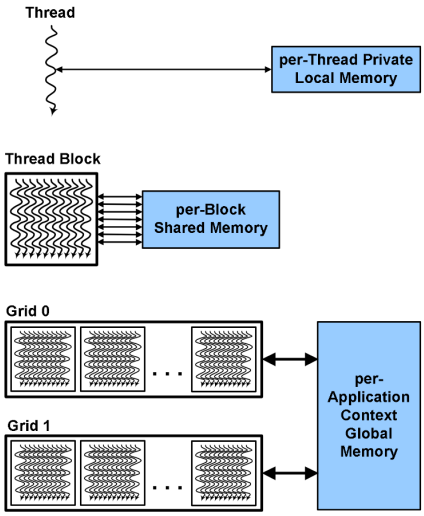


Figure 2.8: Hierarchy of threads, blocks, and grids on parallel computations (from [20]).

The representation of a parallel program, written CUDA or OpenCL, is depicted in the figure 2.8. When creating such a program, the programmer first defines which kernels should be executed. Each kernel is executed in parallel by several threads that the programmer can configure.

The threads, in their turn, are organized in blocks. Usually, thousands of threads are used per kernel execution. Each thread has an ID within its block and access to its registers and private memory. Threads belonging to the same block can cooperate using shared memory and barrier synchronization.

Finally, the blocks themselves are organized in grids. Each grid is composed of all the blocks executing the same kernel and allows both access to global memory and synchronization between dependent kernel calls [20].

2.2.2 GPGPU Architecture

The basic computational building block of a GPGPU is called Streaming Multiprocessor (SM) or Compute Unit (CU) when manufactured by Nvidia or AMD, respectively. Each GPGPU device has at least one of these, whose responsibility is processing a kernel's workload. More precisely, given a workload composed of a given amount of threads organized in blocks, the GPGPU has to schedule such blocks to SMs or CUs. The same SM or CUs can be used for multiple blocks' execution as long as it has the necessary resources to do so. Hence, the maximum number of different kernels being executed at the same time is limited to the number of SMs or CUs a given GPGPU has.

Usually, there are no resources to handle all the blocks at once, which means some blocks have to wait for others to complete their execution before being assigned a SM or a CUs. Furthermore, blocks are assigned to these processors in order, which means all blocks from a given grid are already assigned before another grid's block can be.

Inside the SM, threads are organized in warps, particularly, each warp is commonly composed of 32 threads in current Nvidia's microarchitectures. On the other hand, inside the CU, threads are organized into wavefronts instead, each composed of 64 threads on the Graphics Cores Next (GCN) architecture [21]. Both the warps and wavefronts are executed by Single Instruction, Multiple Data (SIMD) stream processors that the SM and the CU are equipped with.

2.2.2.A Nvidia

Nvidia's Fermi microarchitecture, depicted in figure 2.9, was launched back in 2010 but allows further understanding of the concepts explained above.

Nvidia's Fermi microarchiteture GPGPU device is equipped with 16 SMs organized into 4 Graphic Processing Clusters (GPCs). The device has an engine called GigaThread responsible for scheduling blocks into SMs.

Inside the SM, the warp scheduler and the instruction dispatch unit are responsible for scheduling and issuing each warp's instructions, and since there are two sets available in this architecture, it is possible to execute two warps concurrently within the same SM. The issued instruction for each warp can then make use of 16 out of the 32 stream processors, also known as CUDA cores, the 16 load/store units, or the four 4 Special Processing Unit (SPU). The warp scheduler relies on a scoreboard that analyzes each warp's dependencies to check which ones are ready for execution.

CUDA cores have a pipelined Floating Point (FP) unit and an integer Arithmetic Logic Unit (ALU). On the other hand, SPUs allow the execution of other kinds of instructions such as sin, cosine, reciprocal, and square root. Since there are only four units available, and one instruction is executed per clock cycle, a full warp takes eight cycles to complete within the SPU. To further improve performance, SPU are decoupled from the dispatch unit so that other instructions might be issued while it is occupied [20].

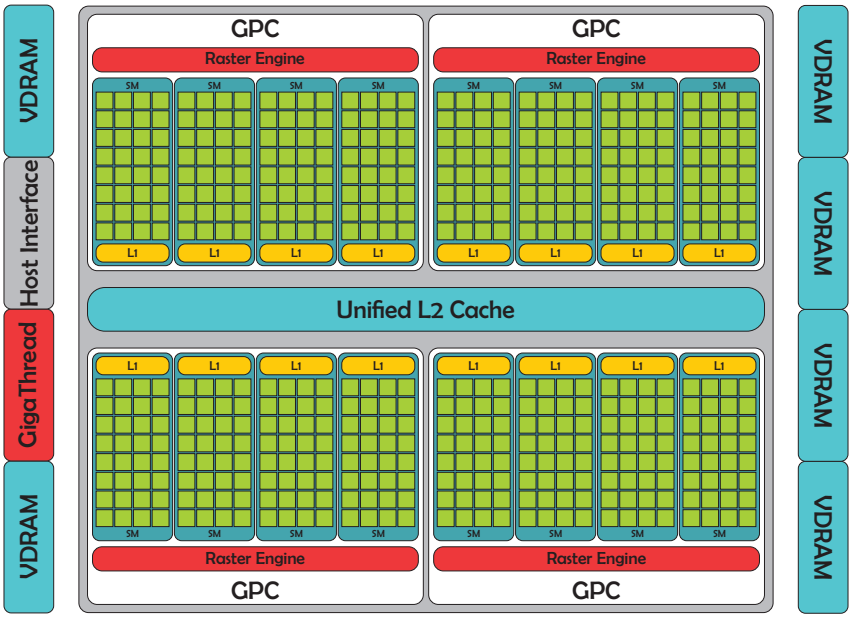


Figure 2.9: Nvidia's Fermi microarchitecture.

A – Tensor Core Nvidia announced in 2017 a microarchitecture named Volta heavily focused on deep learning. It provided optimizations concerning the development of such applications.

The Volta microarchitecture has a redesigned SM architecture and a new processing unit called the tensor core, both optimized for deep learning algorithms [22]. A tensor core is a programmable matrix whose operation is depicted in Equation 2.3. It is located in the SM along with CUDA cores and load/store units. Tensor cores, applied in the first Nvidia GPGPU to support this architecture, Tesla V100, deliver up to 125 TFLOPS, equivalent to a 12 times boost compared to the previous microarchitecture flagship, Nvidia Tesla P100. Both the training and inference phases of a neural network benefit from this unit, as they require, as mentioned, intensive matrix multiplications.

$$D = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} \times \begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{bmatrix} + \begin{bmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{bmatrix} \tag{2.3}$$

A tensor core performs the operation $D = A \times B + C$ as described by the equation 2.3. In a multiplication operation where both operands have the same precision, the result has to have twice as much precision to guarantee no precision loss. With this in mind, matrixes A and B only need to support FP16 (Floating Point represented with 16 bits) to provide a result, matrix D, with FP32 precision. Whereas matrix C has to support FP32 precision operands.

CUDA and also other libraries implemented with it, such as cuDNN and cuBLAS, which were up-

dated to take advantage of tensor cores. Since many deep learning frameworks use these libraries to support their GPGPU acceleration, such as Caffe2 and Tensorflow, they directly benefit from these improvements.

2.2.2.B AMD

AMD's Vega microarchitecture, launched in 2017, is depicted in figure 2.10. It features 64 Next-generation CU, each with a total of 64 stream processors as mentioned above. Similarly to the Nvidia's clusters, GPC, the CU are organized into compute engines.

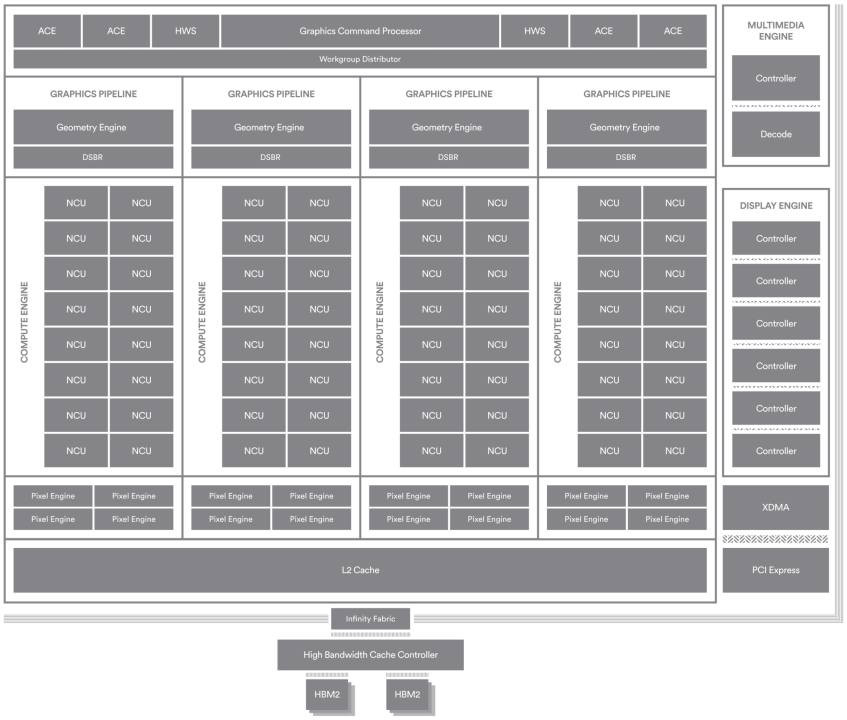


Figure 2.10: AMD's Vega microarchitecture (from [23]).

The design goals of the Vega architecture include power efficiency improvements and scalable performance. One of the ways it achieves that is by shipping new power management microcontrollers that allow implementing more sophisticated power-control algorithms.

Furthermore, the Vega microarchitecture delivers higher frequency clocks when compared to its predecessor. Achieving this metric required redesigned the chip to meet the required timing targets [23].

A – Rapid Packed Math Processing units are vastly optimized for 32-bit FP operations. However, some specialized applications, including deep learning, video processing, and computer vision, do not always require this precision level. Thus, computing 32-bit FP in such cases translates into an overhead.

With this issue in mind, AMD added the Rapid Packed Math feature in the Vega microarchitecture, which adds a new set of 16-bit FP and integer instructions into the instruction set. This set of instructions allows doubling the throughput when compared to their 32-bit counterparts. Naturally, using lower precision calculations also requires less register space and bandwidth.

B – Adaptive Frequency and Voltage Scaling AMD's Vega microarchitecture includes a feature called Adaptive Frequency and Voltage Scaling that targets performance efficiency, which was introduced in the previous microarchitecture named Polaris. The GPGPU device is equipped with sensors responsible for choosing the device's supply voltage and operating frequency. This technique improves the fact that there is a conservative design approach, leading to higher voltage guardbands, and consequently sacrificing efficiency.

To achieve the proposed goal, the GPGPU has a specific circuit that mimics the device's critical path under constant monitoring. Allied with a power supply monitor, that evaluates the available voltage at different regions of the chip, Adaptive Frequency, and Voltage Scaling can measure and intelligently choose the best supply voltage and operating frequency.

After the device boot's up both monitoring systems are used to choose the device's supply voltage and operating frequency. This also has the advantage of accounting for other factors such as aging. The material used to build the GPGPU tends to deteriorate as time goes by, but having this system allows updating the device's supply voltage and frequency accounting for those phenomena [24].

2.2.3 GPGPU accelerated libraries

CUDA and OpenCL, as mentioned, enable the development of low-level parallel kernels in GPGPU devices, by providing an API that a programmer can use within a project. In truth, the programmer does not need to know the actual implementation behind those libraries and still be assured that he is using optimized and efficient primitives. Nonetheless, the programmer might not use those primitives efficiently.

In this subsection, the focus is on libraries that are currently used in deep learning implementations.

2.2.3.A cuDNN

Nvidia provides a library, called cuDNN, which extends their low-level GPGPU library, CUDA, to provide efficient neural networks algorithms accelerated by a GPU. Deep learning algorithms such as activation functions, pooling operations, convolutions, among others are provided in this library whose main intention is to reduce the need for parallel code within deep learning implementations. In fact, it is possible to deploy such programs without writing any parallel code, using only these libraries' functions [25].

Currently, there is no other library providing the same set of GPGPU accelerated primitives as cuDNN provides. Nonetheless, it is possible to use CUDA or OpenCL alone, or even some BLAS library, to build custom primitives as needed [25].

2.2.3.B cuBLAS

Basic Linear Algebra Sub-routine (BLAS) is a set of linear algebra primitives that specify operations such as vector addition, scalar multiplication, dot products, linear combinations, or matrix multiplications.

Nvidia is also in the vanguard of this libraries' implementation featuring GPGPU acceleration. They provide cuBLAS, which also uses CUDA to deploy efficient kernel executions into the GPGPU. This library can relieve intensive algebra computations by distributing them to a single or a cluster of Nvidia GPGPUs.

However, there are more BLAS libraries available. For instance, MAGMA, which allows the exploit of heterogeneous systems, with both CPU or GPGPU. MAGMA has the advantage of providing both OpenCL and CUDA implementations of its library so that any given GPGPU device can benefit from it. Besides, MAGMA also provides specialized libraries for embedded computing [26].

There are also BLAS libraries without GPGPU acceleration, i.e., using only the CPU, such as Eigen, openBLAS, and LAPACK.

2.2.3.C ROCm

Radeon Open Ecosystem (ROCm) is an open-source ecosystem, built by AMD that supports the deployment of scalable systems for high-performance computing and machine learning applications. It includes a collection of libraries, development tools, APIs, drivers, and monitoring tools that support AMD's and other GPGPU devices. Given the open-source nature of the project, existing deep learning frameworks, such as TensorFlow and PyTorch, are integrating ROCm.

At the core of the ROCm ecosystem is Heterogeneous Interface for Portability (HIP), which allows developers to create code ready to compile either into Nvidia's or AMD's GPGPUs. It does do by leveraging Nvidia's proprietary libraries or ROCm libraries themselves, respectively [27].

Similarly to Nvidia's libraries discussed above, ROCm, also includes BLAS libraries through rocBLAS and deep learning libraries through MIOpen [28].

2.3 Computing power performance

GPGPU devices are the main accelerators for deep learning models, and it is widely used across such applications. At the same time, there is a high demand for performance improvements on those applications. However, the performance enhancement of deep learning applications is met with increasingly

higher computational requirements. Consequently, the improvements in the deep learning field can outpace the improvements on the GPGPU devices.

As an example, current computer vision applications are expected to require more than 2 weeks of training on a single GPGPU device. Therefore, researchers opt for large clusters of GPGPU devices to reduce those timings. However, the performance success of this approach depends heavily on an efficient rate of resource utilization and low communication overhead. The same is true for inference, whose challenge is guaranteeing a low latency whilst simultaneously keeping an efficient rate of resource utilization and throughput.

The literature on deep learning improvement techniques for GPGPU devices is rapidly growing. However, the majority of these works neglect the power consumption impact of the proposed solutions. The increasing power consumption is a major challenge on the GPGPU device's usage, especially when considering resource-constrained environments. There is a need for performance improvements to be weighed against their energy overhead, and for a deeper dive on lower precision techniques that impose a trade-off between accuracy and efficiency [2].

2.3.1 Dynamic Voltage and Frequency Scaling Techniques

Dynamic Voltage and Frequency Scaling (DVFS) is a technique used to improve a given processing unit power management. It consists of dynamically updating the processing unit working frequency: Reducing the working frequency will consequently trigger a supply voltage reduction.

Equation 2.4 translates the relationship between the processing unit's frequency f, supply voltage V, and power consumption P.

$$P \propto f \times V^2 \tag{2.4}$$

Since the processing unit's frequency is proportional to its supply voltage, the relationship on the equation 2.4 strongly encourages DVFS when seeking energy efficiency.

First of all, note that DVFS is transversal to any processing unit, including both CPUs and GPGPU. Secondly, reducing the operating frequency has the advantage of reducing the supply voltage and consequently improving the overall energy consumption. However, this is possible at the expense of execution performance, since a lower frequency leads to longer execution timings [29].

Jiao et al. [30] studied how frequency scaling impacts the performance and power consumption of a GTX 280 GPGPU. To achieve that, they used distinct sets of applications categorized into three different groups: compute-intensive, memory-intensive, and hybrid. They observed that the DVFS impacts on energy efficiency are dependent on the application itself. More specifically, it is dependent on

the relationship between global memory transactions and computation instructions. Finally, based on the application properties, both the memory and the cores frequency of the GPGPU device would be adjusted.

Guerreiro et al. [31] proposed a new model to evaluate the potential performance and power consumption improvements of applying DVFS to a given application. They used the profiling result of a set of synthetic benchmarks to train a machine learning model classifier. With the trained classifier it is possible to characterize any kind of GPGPU application performance-wise and power consumption-wise when submitted to DVFS. Their results show that the classifier can successfully predict the optimal frequency for each application, obtaining an average energy saving improvement of $16\,\%$ with a maximum of $36\,\%$.

2.3.2 Voltage operating limit

Even though DVFS techniques do achieve high energy saving potentials, its implementations focus mostly on frequency scaling to achieve energy savings. The voltage reduction is usually a byproduct of the frequency scaling since lower frequencies require lower supply voltages. In other words, the device's supply voltage, within common DVFS implementations, is not actively managed.

However, processing unit manufacturers when designing a processor must account for phenomenons that might negatively impact its performance. When doing so, they usually add a voltage guardband on top of the minimum supply voltage required for the device to function properly. It is estimated that the voltage guardband is approximately $20\,\%$ of the recommended voltage by the manufacturer, i.e. $20\,\%$ of the nominal voltage [3, 4].

Leng et al. [3] study the benefits of exploiting the voltage guardband using a set of commercial GPGPU devices from Nvidia. Their results show energy-saving results ranging up to $25\,\%$. Furthermore, they conclude that the GPGPU device's minimum operating voltage is dependent on the running application.

Additionally, based on each application's performance counters they build a model to predict a given application's minimum operating voltage. The prediction error of their model ranges up to $3\,\%$ with an average of $0.5\,\%$. As the authors suggest, their accurate model opens up possibilities to a dynamic voltage guardband scheme with potential energy savings ahead.

2.4 Summary

This chapter provided a brief introduction to deep learning. It discussed deep learning models, provided an insight into the different layers currently available and which frameworks developers can use to deploy this technology.

GPGPU devices are the main accelerators of deep learning applications. Thus, this chapter also explained how they work behind the scenes, exemplifying with architectures from Nvidia and AMD.

Finally, the computing performance of GPGPU devices was discussed along with existing DVFS techniques used to improve it.

3

Voltage guardband characterization

Contents

The previous chapter laid down the established knowledge regarding deep learning, how GPGPU devices are important accelerators of that technology, and how GPGPUs themselves are designed. Additionally, issues regarding the scalability of deep learning applications under resource-constrained environments were discussed along with potential solutions.

This chapter focus on characterizing an AMD Radeon Vega Frontier Edition GPGPU under a progressively lower supply voltage, effectively forcing the reduction of the voltage guardband imposed by the device's manufacturer. To do so, a set of benchmarks, characterized by their focus on a specific architecture area of the GPGPU, are used.

The choice of an AMD device is twofold. On one hand, the literature regarding DVFS techniques applied to AMD devices is scarce when compared to Nvidia, whilst, on the other hand, AMD does provide the required tools to proceed with this endeavor.

The goal is to identify the minimum working voltage for each benchmark, i.e. the lowest supply voltage that still produces the correct output result. Additionally, this chapter will identify what factors might have an impact on the voltage reduction results, namely frequency, temperature, aging, process variation, and voltage noise. Finally, provides an evaluation of the potential energy savings that reducing the voltage guardband produces.

3.1 Data acquisition

3.1.1 V_{min} and V_{crash}

Voltage guardband is the difference between nominal voltage and V_{min}, the lowest voltage at the GPGPU terminals that allows an application to run correctly. Further reducing the supply voltage below V_{min} will lead to incorrect results or the device's failure.

The correctness of the applications' execution is here defined as the one whose results exactly match the results obtained from the same execution at nominal voltage. Hence, the nominal voltage execution is used as a reference towards the remaining executions, since it is assumed that using the manufacturer's recommended settings will produce the correct result.

Nevertheless, the voltage can be reduced even further allowing errors to happen, and thus affecting the execution correctness. Naturally, this extra reduction is finite leading to the definition of V_{crash}: the voltage value at which the application is no longer executed. There are multiple types of error observed when the device's voltage is reduced below V_{min}, such as Silent Data Corruption (SDC), run-time errors, system crashes, and indefinitely long executions.

SDC occurs when the execution finishes and no warning or error message is triggered, but the final result is not correct, i.e. apparently no error has occurred but the device wrote erroneous data into memory [32]. Run-time errors, on the other hand, occur when the program execution fails during

run-time due to memory access faults and such failure is logged by the system. System crashes and indefinitely long executions require a manual system reboot so that the normal GPGPU device access can be restored.

All programs are executed three times at all voltage levels ranging from the nominal voltage and $30\,\%$ below the nominal voltage, with a resolution of $6.25\,\mathrm{mV}$ as explained in the following sections. Due to the mentioned potential errors which are more prominent below V_{min}, each execution is done after a system reboot.

Rebooting the system before each program execution largely increases the experimental period, but ensures that the system attempts to execute the program at each voltage level. Furthermore, each execution is preceded by a set of instructions that ensure a known state on the GPGPU device. These instructions are issued using a AMD tool as discussed in section 3.1.4. Naturally, the supply voltage is defined at this stage, but also the device's core frequency and fan control. Allied with the system reboot, this set of instructions establishes a common ground for all executions.

3.1.2 Power Measurements

A voltage reduction at the GPGPU terminals naturally implies variations in consumed energy. To quantify this variation and evaluate its advantages, power measurements were collected during all applications' execution.

The power consumption monitoring tool, gpowerSAMPLER [33], is used to register the power metrics of each application execution. This tool allows sampling the applications with customizable intervals providing not only a report of the energy consumption evolution on the GPGPU device but also an overall report of the average consumed energy.

Additionally, gpowerSAMPLER allows sampling of the GPGPU device's temperature using the same customizable intervals. All applications tested during this work were executed using this tool.

3.1.3 Noise Control

Some variables might affect the quality of the measurements and even distort the V_{min} results, such as the temperature and background activities.

In order to minimize the temperature impact i.e., to stabilize the temperature throughout the experiment, the fan speed was fixed for all sets of applications where the same core frequency was used. Thus, guaranteeing that each program is executed under the same temperature conditions.

Regarding the impact of background activities, no other process was running in the GPGPU during the applications' execution, thus reducing background activities' influence on the results. This is further enforced by the system reboot before each program execution.

3.1.4 GPGPU Card

The tests mentioned in this work were executed using the AMD Radeon Vega Frontier Edition graphics card, whose specifications are listed in Table 3.1. This GPGPU device is featured in this study due to three reasons. First of all, there is a lack of studies characterizing AMD devices' behavior when submitted to such constraints, since the majority of the literature focuses on Nvidia devices. Secondly, and as a consequence of the first reason, the GPGPU card used is one of the AMD flagships currently. Lastly, AMD provides all the required tools to proceed with this endeavor and on top of that is growing within the deep learning field, which will be the focus of the next chapter. AMD has recently launched several open-source tools and frameworks dedicated to deep learning which are relevant to the field [27].

To control the voltage at the GPGPU's terminals, AMD GPU Tool (AGT) is used. This tool is also used to control the fan speed during the experiments and to stabilize the working frequency of the cores and memory. However, AGT allows a wider range of tweaks and customizations of the GPGPU device settings.

The minimum resolution allowed by the AGT tool, regarding the core voltage of the GPGPU device, is 6.25 mV. Therefore, the GPGPU core's voltage was decreased using steps of 6.25 mV as stated above.

Table 3.1: AMD Vega Frontier Edition specifications.

Architecture	Vega
Technology	14 nm
Thermal Design Power (TDP)	300 W
Compute Units	64
Stream Processors	4096
Peak Single Precision (FP32)	13.1 TFLOPs
Peak Double Precision (FP64)	819 GFLOPs
Memory Size	16 GB
Memory Type (GPU)	HBM2
Memory Bandwidth	484 GB s^{-1}

All benchmarks are executed using the 1028.57 MHz frequency level. However, in order to study the frequency impact on V_{min} in the subsection 3.3.2.A, the 1107.69 MHz frequency level is also studied. Note that the higher frequency is always explicitly mentioned throughout this document. In all studies, irrespective of the frequency, voltage is progressively reduced from its nominal value, which varies depending on the chosen frequency, until the GPGPU fails to continue such process.

3.2 Benchmarks

Different types of applications, spanning through three distinct sets, were used to characterize the voltage guardband.

The first set includes synthetic benchmarks, which stress only a given component of the GPGPU architecture. This allows understanding to which degree each architecture component impacts V_{min}. Synthetic benchmarks are detailed in subsection 3.2.1.

The second set includes benchmarks with dependencies. Whilst these benchmarks still stress only a given component of the architecture, the instructions are organized in a way that forces dependencies with previous instructions to happen. Different degrees of dependencies were used, ranging from 1 instruction dependency to 8 instruction dependency for each stressed architecture component. Dependency benchmarks are detailed in the subsection 3.2.2.

Both synthetic and dependency benchmarks are based on previous work targeting Nvidia devices. These benchmarks were adapted from CUDA into HIP, which is a portable programming language that can be built using CUDA or Heterogeneous Compute Compiler (HCC), for Nvidia and AMD devices respectively. In this work, given the chosen graphic card, HIP applications were built using HCC [34,35].

3.2.1 Synthetic benchmarks

The first studied set is composed of synthetic benchmarks. Each of these benchmarks stresses only a given component of the GPGPU architecture.

Benchmarks are named according to the architecture component they focus on. `DRAM`, `L2` and `Shared` refer to the memory unit, corresponding to Dynamic Random-Access Memory (DRAM), L2 cache, and shared memory, respectively. On the other hand, `Int`, `SP`, `DP` and `SFU` refer to the functional unit, corresponding to integer, single precision, double precision, and special function unit operations, respectively.

Algorithm 3.1 depicts the kernel code for `Int`, `SP` and `DP` depending on the value type, `integer`, `float` or `double` respectively, of the template variable named `T`. The template variable is one of the arguments of the benchmark itself, defined on line 1 of the algorithm 3.1.

Thus, algorithm 3.1 is an abstract implementation of `Int`, `SP` and `DP`. Each of these benchmarks will instantiate this implementation with the correct data type they represent.

The variable `COMP_ITERATIONS` used in line 3 of the algorithm 3.1, which defines how many iterations should the benchmark complete, has a different purpose besides the simple iteration count cap. By varying `COMP_ITERATIONS`, one can force different combinations of memory accesses and actual computations which, ultimately, translates into slightly different ways of exciting the GPGPU regarding the same functional component.

The variable `UNROLL_ITERATIONS` acts in the same fashion as `COMP_ITERATIONS`, in the sense that it controls the number of iterations executed by the benchmark. The difference is that `UNROLL_ITERATIONS` is preceded by the `#pragma unroll` compiler optimization.

All the tests, including the dependencies benchmarks depicted in subsection 3.2.2, use the following configuration

Algorithm 3.1 Example of synthetic benchmarks code

```
1 template <class T> __global__ void benchmark(int aux, T * result_device){
2
3     for(int j=0; j<COMP_ITERATIONS; j+=UNROLL_ITERATIONS){
4         #pragma unroll
5         for(int i=0; i<UNROLL_ITERATIONS; i++){
6             r0 = r0 * r0 + r1;
7             r1 = r1 * r1 + r2;
8             r2 = r2 * r2 + r3;
9             r3 = r3 * r3 + r0;
10        }
11    }
12
13    result_device[hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x] = r0;
14 }
```

$$\texttt{COMP\_ITERATIONS} = 1024 \tag{3.1}$$

$$\texttt{UNROLL\_ITERATIONS} = 32. \tag{3.2}$$

Having the same configuration ensures all benchmarks execute the same number of operations, thus guaranteeing consistency across all the benchmarks.

The `#pragma unroll` optimization tag, used in line 4 of algorithm 3.1, is also worth further detailing. It is, as mentioned, a compiler optimization which, as the name suggests, unrolls a loop. An example of this behavior is presented in algorithms 3.2 and 3.3. By doing this optimization, the compiler will translate each of the loop's iterations into explicit instructions as exemplified in algorithm 3.3, i.e unroll the loop.

More precisely, and using the algorithm 3.2 as an example, apart from assigning the value of the variable `index` to the vector `vector`, by not unrolling the loop the processor would need to initialize the variable `index`, increase the variable `index` 8 times and evaluate 9 times if `index` is less than 8.

Another advantage of loop unrolling lies in the enhancement of Instruction-Level Parallelism, since GPGPU's execution incurs a performance penalty for every branch it executes.

Algorithm 3.2 Simple loop

```
1 for (int index = 0; index < 8; index++ )
2     vector[index] = index;
```

Even though the algorithm 3.1 does not show any evidence of that, all the benchmarks were initialized

Algorithm 3.3 Compilation result of algorithm 3.2 using `#pragma unroll`

```
1 vector[0] = 0;
2 vector[1] = 1;
3 vector[2] = 2;
4 vector[3] = 3;
5 vector[4] = 4;
6 vector[5] = 5;
7 vector[6] = 6;
8 vector[7] = 7;
```

with a known value allowing for the extraction of the benchmark final result. The benchmark final result is used to evaluate its execution correctness when compared against the same value under normal conditions, i.e. at the GPGPU nominal voltage.

3.2.2 Dependencies benchmarks

Dependency benchmarks extend the previous synthetic benchmarks. Thus, dependencies benchmarks also have the same architecture-specific type of application with a focus on functional architecture. However, a new element is introduced: dependencies.

The execution of a given program is described by the sequential processing of each of its instructions by the processing unit. Since the execution is sequential, when the program loads a new instruction it is assumed that the previous ones are completed and their results are available. However, due to the architecture of modern processing units, some instruction's result might not be ready by the time a new instruction is trying to fetch it.

This means the processing unit must either stall the execution until the result is ready, actively push data from the previous instruction execution stage or a combination of both [36, 37].

Within the realm of dependencies, there are different types of dependencies: control and data dependencies, depending on whether the execution is compromised due to control instructions or data instructions, respectively. Control dependencies occur when the execution of an instruction is decided by the evaluation of a previous one. On the other hand, data dependencies occur when contiguous instructions share the same resources e.g., registers.

Algorithm 3.4 exemplifies what a control dependency looks like. The execution of the last instruction depends on the result of the first one. Thus, that instruction is control dependant on the first one. The processing unit must solve this dependency to guarantee the program's correctness, considering that the following statements might use the same resource `R2`.

Furthermore, data dependencies can be torn down into different subgroups. More specifically, Read-after-Write (RAW), Write-after-Read (WAR), and Write-after-Write (WAW) dependencies, which are ex-

Algorithm 3.4 Control Dependency

```
1 IF (R1 = A)
2     R2 = B
```

emplified in the algorithms 3.5, 3.6 and 3.7 respectively.

The names of these data dependencies are self-explanatory. RAW dependency, or flow dependency, occurs when the processing unit is trying to read a value that hasn't yet been written in previous instructions. This is what happens in the algorithm 3.5: The value `A` might not be available in `R1` by the time the processing unit starts executing that last instruction.

Algorithm 3.5 RAW - Flow Dependency

```
1 R1 = A
2 R2 = R1
```

Similarly, a WAR dependency, also known as an anti-dependency, happens when the processing unit reads from a resource that is updated afterward. In the algorithm 3.6, the variable `R1` is updated immediately after its value being read.

This dependency is also known as a *name dependency* because using another processor resource would solve the issue. In the example, using any other variable besides `R1` would be innocuous.

Algorithm 3.6 WAR - Anti-dependency

```
1 R2 = R1
2 R1 = A
```

Finally, a WAW dependency arises when a program as two contiguous instructions updating the same resource. As the WAR dependency, this one is also a *name dependency* due to the same reasons.

Plugging this knowledge into the synthetic benchmarks described in the previous section creates the dependencies benchmarks. As mentioned, processing units have the means to solve these issues as a given program is executed. Nonetheless, the processing flow suffers an overhead whilst solving them. The goal with this set of benchmarks is to characterize the processing unit's voltage guardband under the known overhead these benchmarks introduce.

Considering the nature of each of the previous data dependencies, this section solely focuses on RAW dependencies, which are also known as *true dependencies*. This is because *name dependencies*, which include WAR and WAW, are easily solved by the compiler itself.

Algorithm 3.8 illustrates the data dependency kernel used in this study. The core kernel instructions,

Algorithm 3.7 WAW - Output Dependency

```
1 R1 = A
2 R2 = R1
3 R1 = B
```

from line 6 through line 9, all depend on the result of the previous one i.e., one instruction dependency, thus forcing the RAW data dependency.

To evaluate the degree to which these hazards influence the execution, variants of the illustrated kernel are created. Variants are created by varying the number of instructions between the dependent ones. As an example, a two instruction dependency means that the current instruction depends on the result of the second last instruction. Similarly, a three instruction dependency means that the current instruction depends on the result of the third last instruction. Dependency benchmarks are named in the same manner as the synthetic ones with a suffix stating the number of instructions the kernel depends on.

Algorithm 3.8 Dependency benchmark kernel (1 instruction dependency)

```
1  template <class T> __global__ void benchmark(int aux, T * result_device){
2
3      for(int j=0; j<COMP_ITERATIONS; j+=UNROLL_ITERATIONS){
4          #pragma unroll
5          for(int i=0; i<UNROLL_ITERATIONS; i++){
6              r0 = r0 * r0 + r3;
7              r1 = r1 * r1 + r0;
8              r2 = r2 * r2 + r1;
9              r3 = r3 * r3 + r2;
10         }
11     }
12
13     result_device[hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x] = r0;
14 }
```

This code is also compiled using the AMD HCC. Then, each benchmark variant is executed for the three frequency levels and all the voltages ranging from the frequency nominal voltage until the GPGPU device failure with 65 mV decrements.

Energy consumption and execution time metrics are extracted from each execution using the tools mentioned in previous sections. Doing so allows characterizing the GPGPU device under the dependencies constraints.

3.3 Evaluation

3.3.1 Minimum operating voltage

3.3.1.A Synthetic Benchmarks

Figure 3.1 plots the measured V_{min} for the studied synthetic benchmarks. Further reducing the voltage below V_{min} results in either run-time errors, system crashes, or indefinitely long executions. In other words SDC did not occur when the device's voltage was below V_{min}. Thus, V_{crash} matches V_{min} results for the synthetic benchmarks.

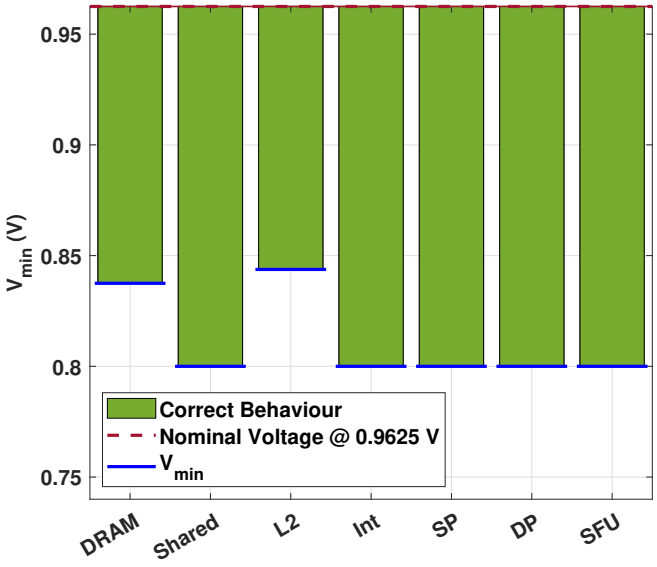


Figure 3.1: Obtained V_{min} for synthetic benchmarks.

The first empirical conclusion is that multiple V_{min} results were obtained across the different benchmarks. The group of benchmarks that target the ALU, which includes `Int`, `SP`, `DP` and `SFU`, all obtained along with `Shared` a V_{min} of 0.8 V at 1028.57 MHz. `DRAM` and `L2` benchmarks obtained a V_{min} of 0.8375 V and 0.843 75 V, respectively.

Despite the synthetic benchmarks being a small set of programs and that some V_{min} clusters can be defined nonetheless, there are different V_{min} results across different applications. This is aligned with the expectations that V_{min} depends mostly on the application itself [3]. Section 3.3.2 focus on exploring this the cause of the V_{min} variability.

It is important to note that the DRAM component of the GPGPU plays an important role across all benchmarks. All benchmarks rely on the DRAM memory to communicate back with the CPU. This communication is crucial so that it is possible to evaluate the correctness of the benchmark result.

What distinguishes the `DRAM` benchmark from the remaining ones is that `DRAM` executes multiple

writes to the DRAM memory whilst the other benchmarks only write in that memory at the end of their execution. Given the results of the `DRAM`, benchmark it is also possible to conclude that writing to the DRAM has a high impact on V_{min}.

Further analyzing the results presented in figure 3.1, one can infer that a significant voltage guard-band is obtained for all the benchmarks ranging from $12.34\,\%$ to $16.88\,\%$ with an average of $15.68\,\%$ regarding nominal voltage of $0.9625\,\text{V}$. These values are slightly below the $20\,\%$ range obtained in previous works [3, 38, 39].

3.3.1.B Dependencies benchmarks

The obtained results for benchmarks with dependencies are plotted in figure 3.2. Naturally, as previously mentioned, no memory benchmarks are present since data dependencies cannot be forced on memory-only applications.

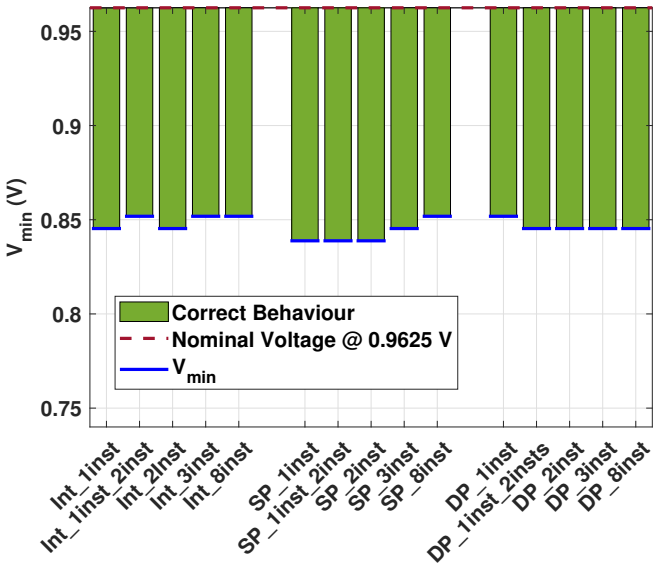


Figure 3.2: Obtained V_{min} for benchmarks with dependencies.

Hence, the considered set of benchmarks include `Int`, `SP` and `DP` applications, that is integer, single-precision and double-precision only applications, respectively. There are also 4 types of data dependencies for each application. Each of these data dependency types is identified by suffixing the name of the application with `1_inst`, `1_inst_2_inst`, `3_inst` and `8_inst`. These suffixes are self-explanatory. For instance, `1_inst` means each issued instruction has a dependency with the last issued instruction, i.e. 1 instruction dependency. Analogously, `8_inst` means each issued instruction has a dependency with the eighth last issued instruction, i.e. 8 instructions dependency. `1_inst_2_inst` means there is simultaneously a dependency with the last and the second last issued instruction.

Empirically, a larger adjacency between dependent instructions implies a higher computational cost, since more stalls have to be introduced by the processor to guarantee the correct result is available for a given instruction's operand before it is issued. With that in mind, it is expected for V_{min} to be lower on `1_inst` suffixed applications, when compared with `8_inst` suffixed applications, as an example. However, that conclusion cannot be inferred from the results presented in figure 3.2 since mixed results are obtained: for `DP` instructions it is observable the exemplified expected behavior. Yet, for `SP` instructions, the opposite is observed.

Additionally, one other thing to note is that similarly to the synthetic benchmarks, dependencies benchmarks' execution did not lead to incorrect results, meaning that V_{crash} is equal to V_{min} in all these applications.

Finally, even though the V_{min} variation is not as steep as expected within the dependencies benchmark, when these are compared, as a whole, to the synthetic benchmarks one can observe a few differences. For `Int`, `SP` and `DP`, V_{min} is on average $0.853\,75\,\text{V}$, $0.8475\,\text{V}$ and $0.851\,25\,\text{V}$ on the dependencies benchmarks, respectively. These values correspond to an undervoltage of $11.30\,\%$, $11.95\,\%$ and $11.55\,\%$. The obtained undervoltage for the dependency benchmarks is lower than the one benchmarks obtained for the synthetic benchmarks. This result is aligned with the expectations that dependency benchmarks would produce a lower voltage guardband optimization when compared to the synthetic benchmarks.

3.3.2 Impacts of voltage guardband and frequency optimizations

The previously presented experimental results show that V_{min} varies depending on the program that is executed. More specifically, results show a V_{min} variation ranging from $0.8\,\text{V}$ to $0.86\,\text{V}$, corresponding to a variability of $0.06\,\text{V}$ or $6.2\,\%$ when compared to the nominal voltage.

The obtained $0.06\,\text{V}$ variability is $40\,\%$ lower than the variability of $0.1\,\text{V}$ obtained by Leng et al. [3]. This difference can be explained by the different frequency settings used and the set of benchmarks used in the current study.

First of all, this study features a smaller set of benchmarks where each of them focuses on a specific GPGPU architecture unit of the processing device. Thus, these benchmarks are, by nature, less complex and less resource-dependent. Contrasting these properties with the broader set of programs used in the mentioned work explains the variability difference. Furthermore, the results discussed in chapter 4 prove the variability increase as the complexity of programs analyzed also increases.

Despite results showing a lower V_{min} variability, different programs indeed showed different V_{min} results. The goal of this subsection is to study which variables are the main contributors to such variability. The discussed variables include operating frequency, temperature, aging, process variation, and voltage noise.

3.3.2.A Operating Frequency

To evaluate the impact the frequency has upon V_{min} variability, the voltage guardband experiment described previously was repeated under different constraints. All the controlled variables, such as fan speed, were kept and frequency was increased to a higher level using AGT.

Note that AGT allows manipulating two different frequencies of the GPGPU device: memory frequency and core frequency. As the names suggest, these properties are used to set the frequency at which the GPGPU dedicated memory and cores are working on, respectively. In this study, the focus is on the latter: the cores frequency. This is because those cores are responsible for the actual program processing.

Figure 3.3 shows the obtained V_{min} readings at the two studied frequency rates: 1028.57 MHz and 1107.69 MHz.

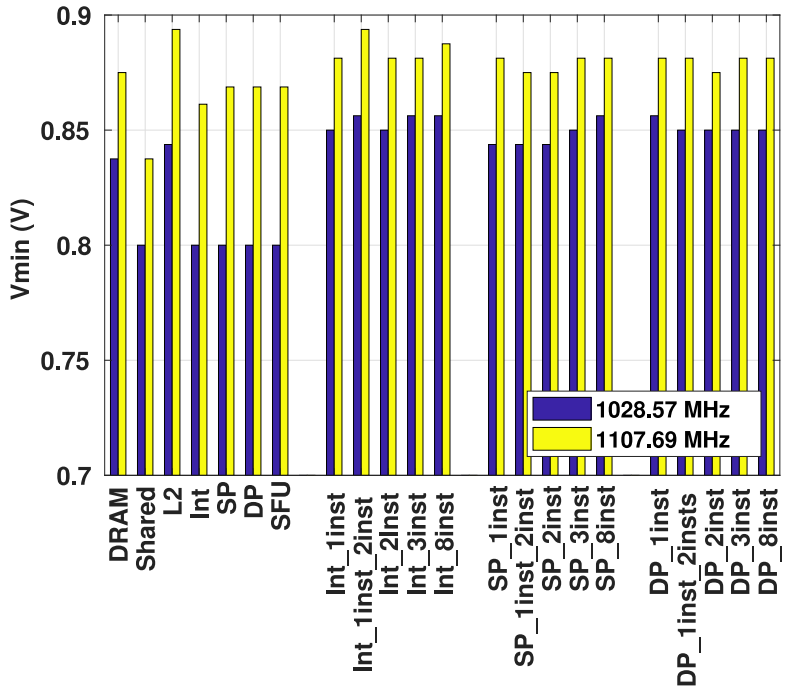


Figure 3.3: V_{min} obtained for the benchmarks at 1028.57 MHz and 1107.69 MHz.

On an empirical analysis, benchmarks that attained a higher V_{min} reading at 1028.57 MHz, also obtained a higher V_{min} result at 1107.69 MHz. Once again, it is noticeable that synthetic benchmarks achieve lower V_{min} results when compared to their dependency counterpart at both core frequencies.

Additionally, the V_{min} variability on the higher frequency experiment matches exactly the variability of the lower frequency experiment. The difference between the minimum and maximum obtained V_{min}

is 0.06 V in both cases.

Furthermore, one can observe the considerably higher V_{min} reading of the higher frequency, at 1107.69 MHz. Whilst that is true, it is also true that the GPGPU device requires a higher nominal voltage to operate at a higher frequency level. Nominal voltage is defined as the out of the box voltage setting, established by the manufacturer. Specifically, to the AMD Radeon Vega Frontier featured in this study, its nominal voltage is 0.9625 V when its cores are working at 1028.57 MHz and 1.0375 V when its cores are working at 1107.69 MHz.

Thus, to establish a common ground of comparison between both frequency experiments, V_{min} results were normalized against the nominal voltage using equation 3.3. Note that, as mentioned above, $V_{nominal}$ is not the same at different operating frequencies. The goal of normalizing V_{min} is to evaluate the real impact of the GPGPU core frequency on the device's voltage guardband.

$$V_{normalized} = \frac{V_{min}}{V_{nominal}} \tag{3.3}$$

The figure 3.4 plots V_{min} normalized for the lower frequency vs V_{min} normalized for the higher frequency for all benchmarks. In other words, each point in the plot represents one benchmark: its normalized V_{min} at 1028.57 MHz dictates its position on the vertical axis, whilst the normalized V_{min} at 1107.69 MHz dictates its position on the horizontal axis.

Furthermore, the figure 3.4 also includes two dashed lines. The first assumes a scenario the voltage guardband is proportional to $V_{nominal}$ on both frequencies. If this is the case, each benchmark will obtain the same normalized V_{min} when the GPGPU is working at different frequencies, i.e. $V_{min@1028.57\,MHz} = V_{min@1107.69\,MHz}$. The frequency impact on the voltage guardband can be measured from how far away from this dashed line the actual results are. Consequently, the second dashed line represents the linear regression of all points in the graph. It allows evaluating how far from the first dashed line the results are on average, effectively measuring the frequency impact on the voltage guardband.

The linear regression from the data points extracted from figure 3.4 is defined on equation 3.4. There is an observable proclivity for higher V_{min} readings on lower core frequency settings, which translates into a lower voltage guardband on lower frequencies.

According to figure 3.4, V_{min} tends to be higher on lower core frequencies, as proven by the linear regression of the data points in the figure, described by equation 3.4 and identified with the label *"Average Impact"* on the figure.

$$V_{min@1028.57\,MHz} = V_{min@1107.69\,MHz} + 0.0258 \tag{3.4}$$

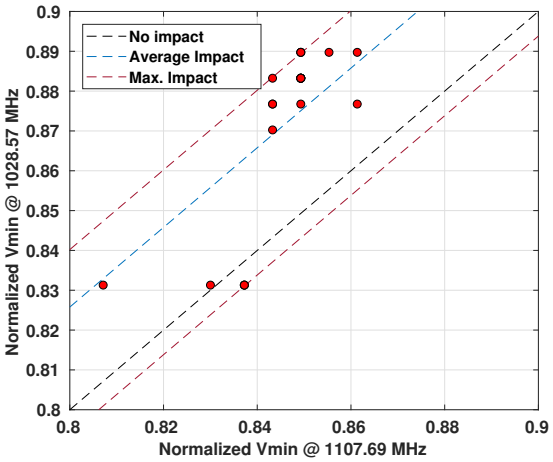


Figure 3.4: Benchmark's normalized V_{min} at 1028.57 MHz plotted against their normalized V_{min} at 1107.69 MHz.

The linear regression defined by equation 3.4 reveals that the lower core frequency, running at 1028.57 MHz, achieves on average 2.58 % higher V_{min} results. This means that the gap between $V_{nominal}$ and V_{min} is negatively impacted by 2.58 % on the higher frequency experiment when compared to the lower frequency counterpart. In other words, the voltage guardband is 2.58 % lower the higher frequency experiment at 1107.69 MHz.

Furthermore, figure 3.4 also contains two dashed lines identified with the label *"Max Impact"*, which are described by equation 3.5 and equation 3.6. These lines, with a unitary slope, correspond to the boundaries of all the data points in figure 3.4, representing the maximum impact the GPGPU device's core frequency had on V_{min}.

$$V_{min@1028.57\,\text{MHz}} = V_{min@1107.69\,\text{MHz}} - 0.0062 \tag{3.5}$$

$$V_{min@1028.57\,\text{MHz}} = V_{min@1107.69\,\text{MHz}} + 0.0402 \tag{3.6}$$

In conclusion, results show that the frequency has virtually no impact on the V_{min} variability since both experiments achieved the same 0.06 V V_{min} variability. However, programs running at a higher frequency show the proclivity to have a lower voltage guardband, i.e. a reduced distance between $V_{nominal}$ and V_{min}. Nonetheless, this tendency is a negligible 2.58 %.

3.3.2.B Temperature

As mentioned, all program executions were encapsulated within gpowerSAMPLER which allows measuring a set of metrics, including the GPGPU device temperature.

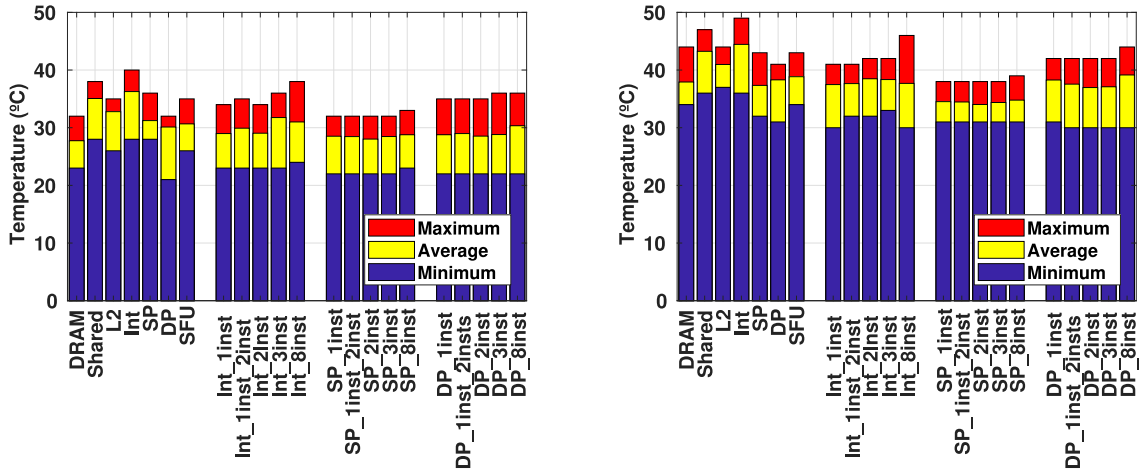
In order to study the temperature impact in V_{min} measurements, both synthetic and dependency,

benchmarks were executed in the same fashion as before and using two different temperature levels at a fixed frequency.

With the available equipment, it is not possible to enforce a steady temperature reading on the device. However, AGT allows controlling the fan speed of the AMD Radeon Vega Frontier. Using this property of the tool one can choose to set the fan speed to low, medium, or high intensity. There is also a fourth setting called auto that, as the name suggests, adjusts the fan intensity dynamically according to the temperature sensor.

In light of the temperature study, benchmarks were executed using both medium and high fan intensity at a fixed frequency. Tests using low fan intensity led the program execution to lag indefinitely and in some cases to system failures, thus preventing tests to reach a wider temperature range.

Figure 3.5 depicts the minimum, average, and maximum temperature reading levels for each of the benchmarks. The figure is composed by two sub-figures, 3.5(a) and 3.5(b), which represent the high and low fan intensity, respectively.



(a) Low-temperature

(b) High-temperature

Figure 3.5: Minimum, average and maximum temperature variation for both synthetic and dependency benchmarks.

The mentioned figure reveals a wide range of temperature levels at the GPGPU throughout the program execution, both on the low and the high-temperature experiments. More specifically, the low-temperature experiment resulted in temperature readings varying, on average, from a minimum of 23.5 °C to a maximum of 34.7 °C, with a global average of 32.9 °C. The high-temperature experiment resulted in temperature readings varying, on average, from a minimum of 31.9 °C to a maximum of 42.0 °C, with a global average of 39.7 °C. The minimum to maximum variation is 31.7 % and 39.3 % respectively.

Despite the limitations of the temperature enforcement method, the obtained results show an average

difference of 6.8 °C between the low-temperature and the high-temperature experiment.

Given the foundation of the experiment and the obtained temperature values, figure 3.6 plots the V_{min} results from the low-temperature experiment against the high-temperature experiment in an attempt to study the temperature's impact on the voltage guardband.

It is also important to note that this study used the same frequency in both low-temperature and high-temperature experiments, thus removing one variable impact.

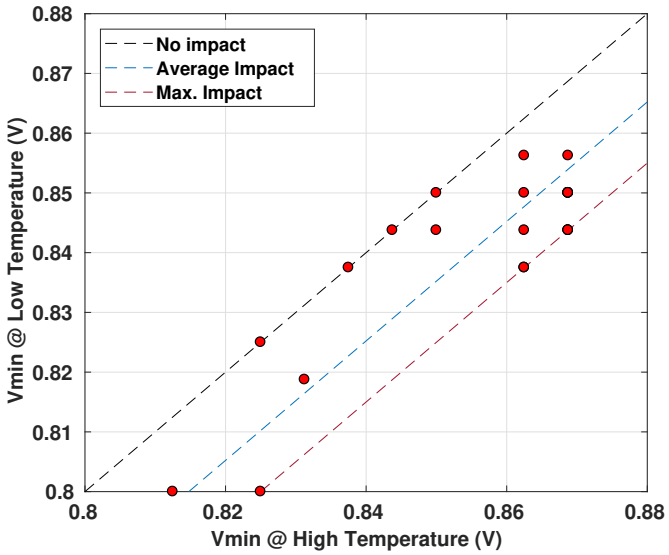


Figure 3.6: Benchmark's V_{min} at different temperatures. The distance between the dashed lines represents the impact of the device's temperature on V_{min}.

According to figure 3.6, V_{min} tends to be higher with higher temperature values. There are three dashed lines plotted along with the V_{min} scattered pairs. Those lines are described by the following equations:

$$V_{min@LowTemperature} = V_{min@HighTemperature} \tag{3.7}$$

$$V_{min@LowTemperature} = V_{min@HighTemperature} - 0.0148 \tag{3.8}$$

$$V_{min@LowTemperature} = V_{min@HighTemperature} - 0.0250 \tag{3.9}$$

Naturally, points plotted coincidentally on equation 3.7 represent those benchmarks where the obtained V_{min} is the same in both the low-temperature and the high-temperature experiments. All V_{min} pairs are plotted at or below that dashed line, which validates that V_{min} tends to be higher with higher temperature values.

Moreover, the second dashed line, described by equation 3.8, is the regression of all the V_{min} pairs,

i.e. the linear approximation that best fits the given data set. From that, one can infer that the temperature has an average impact of 0.0148 V in the V_{min} variability.

Lastly, the third dashed line, described by equation 3.9, corresponds to the linear equation, with a unitary slope, that crosses the data points further away from the first dashed line. Thus, the temperature had a maximum impact of 0.0250 V on V_{min}. More specifically, V_{min} was at most 0.0250 V lower on the low-temperature experiment.

In conclusion, the impact temperature has on the V_{min} variability is not enough to explain the whole magnitude of the V_{min} variability observed. This result is aligned with [3], where the temperature impact was evaluated at 0.02 V which is similar to the current results. The slightly reduced impact obtained here might be related once again to the nature of these benchmarks and also to the extreme temperature levels they were able to test. Due to the limitations explained above, those temperature levels were not possible to achieve here. It is also important to note that this study used the same frequency in both low-temperature and high-temperature experiments.

3.3.2.C Aging

AMD Vega Frontier is built using 12.5 billion 14 nm transistors using Low Power Plus (LPP) FinFET process technology [23]. FinFET is a Metal Oxide Semiconductor (MOS) Field Effect Transistor (FET) which uses a different architecture than the conventional MOS FET. With the increasing reduction of the transistor gate's size, it's effects on the source to drain channel is reduced, thus reducing the transistor performance. The new FinFET architecture solves this problem by wrapping the gate electrode around the source to drain channel [40].

Furthermore, a given MOS transistor can either be a n-type Metal Oxide Semiconductor (NMOS) or a p-type Metal Oxide Semiconductor (PMOS) depending on whether it is built using an n-type or p-type semiconductor. An n-type semiconductor is a dielectric that has been doped with an electron donor dopant, whilst the p-type semiconductor was doped with an electron acceptor dopant. These dopped semiconductors are the key element of modern electronic components and digital circuits.

Now, within the semiconductor industry, there is an issue, known for more than 50 years, called Negative Bias Temperature Instability (NBTI) that negatively impacts the reliability of the PMOS transistor technology [41]. Similarly, there is the Positive Bias Temperature Instability (PBTI) phenomenon counterpart which mainly affects the reliability of the NMOS.

NBTI consists in the accumulation of positive charges at the transistor's gate insulator due to a negative bias voltage, V_g, at the transistor's gate. This process is aggravated by temperature, hence its name. The accumulated positive charges partially cancel out the gate's applied voltage, since the PMOS is activated by a negative electric potential difference at the gate's terminal. Consequently, the source to drain current flow is reduced because the accumulated charges reduce the effect of the gate

and at the same time do not contribute to the transistor conduction channel. Ultimately this phenomenon leads to the transistor's performance loss.

Conversely, PBTI occurs due to a positive bias voltage, V_g at the transistor's gate. In this case, negative charges are accumulated in the gate's insulator which has the same effect of canceling out the gate's applied voltage, since NMOS is activated by a positive electric potential difference at the gate's terminal.

As mentioned, the GPGPU featured in this study, AMD Vega Frontier, uses MOS transistors, which are subjected to NBTI and PBTI effects. Given that transistors are the building blocks of a digital circuit, any effect on the device's building blocks is bound to also have an impact on the device itself. Hence the importance of the NBTI and PBTI.

Regarding the current study, all the test executions on the GPGPU device were completed during a period of six months. This period is considerably low when compared to the typical lifetime of such devices of ten years. Furthermore, some techniques can be applied to slow down the aging processing with minimum overhead [42].

During the period of this study's tests, no shreds of evidence of V_{min} variability, due to the aging process on the GPGPU device, were found. It is also important to note that the impacts of aging are not the main focus of this study.

Furthermore, related work in this field shows a performance impact of up to $2\,\%$ under real-use conditions on International Business Machines Corporation (IBM) microprocessors [43]. In conclusion, it is unlikely that aging factors alone could explain the V_{min} variability observed in this study.

3.3.2.D Process Variation

Production processes are liable to degrees of variation that impact the quality of the final product. The quality of the products themselves can be used to describe the process quality, through which they were produced, using two variables: accuracy and precision.

Figure 3.7 is a conceptualization of these two variables. It assumes a normal distribution of the number of products produced along with a given quality standard.

On one hand, a production process can be on target, but with high variability. This means the production process is accurate. On the other hand, it might be off-target, but consistent. This means the production process is precise.

Furthermore, a production process can be both off target and have high variability. This scenario means it is not accurate nor precise. Finally, the production process can be on target and consistent, which means it is both accurate and precise [44].

Naturally, problems that cause a process to be off-target are easier to identify than problems that cause it to be variable. In order words, accuracy problems are easier to solve than precision problems.

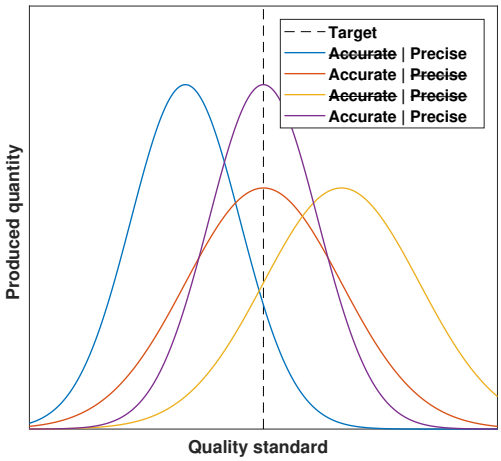


Figure 3.7: Accuracy and precision differences when plotting the outputs of a process against a quality standard.

The transistor manufacturing process is also exposed to such quality variables. Particularly, there are precision issues with the manufacturing process that have been broadly studied by the community. These precision issues are called Process Variation (PV) and refer to the variability of the device's parameters, such as the transistor's gate width, the channel length, or the oxide thickness, from their nominal specifications.

PV has become increasingly more severe due to the increasing difficulty to precisely control the fabrication process as the transistor size became smaller. The chip yield i.e., the fraction of fully working chips within the wafer where they are produced, was reduced from 90 % to 50 % and then 30 % when the transistor size scaled from 350 nm to 90 nm and then to 30 nm respectively [45].

In order to study the PV impact on the V_{min} variability, it is necessary to complete the V_{min} study using multiple GPGPU devices, and evaluate how each of those devices performs. This way, the effects of PV would spread along with the devices and a conclusion can be made in that regard.

Due to constraints, during this study, there was only a single AMD Radeon Vega Frontier Edition GPGPU device available. Thus, an experimental study of the PV effects on the V_{min} could not be conducted.

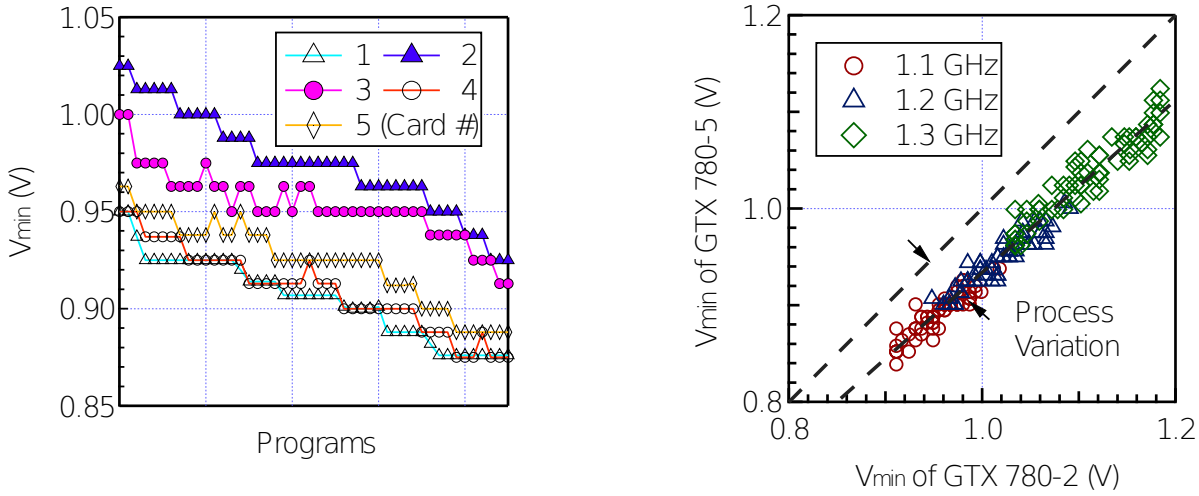
Leng et al. [3] briefly studied the impact of PV in the voltage guardband of Nvidea GPGPU devices. They used only five different Nvidia GTX 780 GPGPU devices, which, as the authors themselves suggest, is not statistically robust. Nonetheless, the results provide insights into the PV effect on such endeavors that are useful to comprehend the root cause of the V_{min} variability.

They observed a constant offset of V_{min} between each of the tested GPGPU devices, with deviations on a few programs as shown by figure 3.8(a). This offset means that a given program experiences V^1_{min} and V^2_{min} on GPGPU device 1 and 2, respectively, but $V^1_{min} \neq V^2_{min}$. More precisely, Leng et al. [3]

observed a 0.07 V maximum offset between the five GPGPU devices. Naturally, this offset is introduced by PV of the GPGPU production process.

The constant V_{min} offset is also depicted in figure 3.8(b), where the V_{min} of two of the five GPGPU devices are plotted against each other. Their results also show a slight increase in the offset as the frequency increases.

On top of the constant offset, it is important to note that the applications they tested also showed random V_{min} deviations from each GPGPU device to the other. Even though each GPGPU device experienced deviations from the constant offset on some programs, those deviations happened in different programs at each device.



(a) Applications' V_{min} results on five Nvidia GTX 780 GPGPU devices (from [3]). A constant and a random offset is observed between each device's V_{min}.

(b) Applications' V_{min} from GPGPU device 2 and 5 plotted against each other at different frequencies (from [3]). PV impact on V_{min} is shown as the difference from the data points to the line $y = x$.

Figure 3.8: Leng et al. [3] results of PV impact on V_{min}.

The observed V_{min} variability is caused by PV, which is known for creating both systemic and random variances on the device's building blocks parameters [46]. Thus, the constant offset can be attributed to the systemic variance imposed by PV, whilst the observed deviations from that offset can be attributed to the random variance imposed by PV.

A slight variation on the digital circuit components has the potential of changing the circuit's critical path. Therefore, programs that do not rely on the critical path in one device, might do rely on it on other devices, thus explaining random V_{min} deviations from the constant offset.

Notwithstanding, there was V_{min} variability on the tested programs across each GPGPU device. Additionally, that variability has approximately the same magnitude, on all devices. However, there were

indeed differences in the absolute values of V_{min}, which can be attributed to PV, but PV itself cannot explain the observed variability.

3.3.2.E Voltage Noise

The voltage signals that powers a digital circuit is not steady as one might expect because there is a degree of fluctuation associated with it. This fluctuation is called voltage noise.

Voltage noise is not desired, except for some circuits with specific goals. Therefore, a considerable drop in the voltage signal can lead to unexpected behavior from the circuit. With this in mind, digital circuit manufacturers increase the supply voltage above the circuit's intrinsic voltage. Thus creating the mentioned voltage guardband which acts as a safe margin usually greater than $20\,\%$. With the increasing scaling of the digital circuit manufacturing process, this margin percentage has the potential to increase [39].

The voltage guardband is also added as a protection against phenomenons such as temperature, aging, PV, and voltage noise. As discussed in the previous subsections temperature, aging and PV does not explain, by themselves, the whole extent of the V_{min} variability. Thus, by exclusion, voltage noise is the main contributor towards the V_{min} variability [3].

Note that, the voltage guardband does not directly include concerns regarding the device's operating frequency. This is because increasing the frequency reduces the available signal propagation time. Consequently, the intrinsic V_{min} has to increase to support those timings. This is the reason why the higher frequency has a higher nominal voltage imposed by the manufacturer.

The voltage at a given point, A, in an electrical circuit is given by the equation 3.10.

$$V_A = V_{DD} - I \cdot R - L \cdot \frac{di}{dt} \tag{3.10}$$

From the equation, 3.10 one can conclude that two factors impact the voltage: the current draw and the current draw's increasing rate.

Leng et al. [3] studied further how each of those factors impacts the voltage at the same point A. Given the equation 3.11 differentiated from Ohm's law, the hypothesis of $I \cdot R$ being the dominant factor in the voltage (equation 3.10) was tested.

$$P = R \cdot I^2 \tag{3.11}$$

If this hypothesis holds, a program with a high power consumption would have a high voltage noise and consequently a higher V_{min}.

Using power consumption measurements and also the Instructions Per Clock (IPC) as a predictor for

power consumption, no evidence of a correlation between those properties and V_{min} was found. Thus, $I \cdot R$ is not the dominant factor of the voltage noise, and consequently, di/dt is.

Leng et al. [3] proceed even further to study which program activity is responsible for generating the most di/dt droop. The study included CUDA runtime activities, inter-kernel activities, initial-kernel activities, and intra-kernel activities. CUDA runtime activities refer to the activities launched by the runtime API to manage the kernels' execution. In the current study, those would be HCC runtime activities.

Their conclusion is that the greatest driver of di/dt droop and consequently the driver o voltage noise is the intra-kernel activities. These are related to the nature of the kernel itself, which varies from application to application. They also identify cache misses and pipeline stalls as a driver for di/dt droop.

Focusing on the current study, it is possible to conclude that the results match the established knowledge. Dependency benchmarks obtained significantly higher V_{min} results when compared to their synthetic counterpart. Whilst recovering from pipeline stalls, the GPGPU moves away from an idle state. Thus, there is a sudden increase in the drawn current which in its turn produces a voltage spike.

3.3.3 Energy gains

With V_{min} values obtained for each application, it is now feasible to evaluate how the energy consumption improves when the voltage is set to those levels at the GPGPU's terminals. Energy savings are measured by comparing the consumed energy when the GPGPU is operating at V_{min} against the same metric when the device is running at the nominal voltage for the selected core frequency: 1028.57 MHz.

With everything set up, gpowerSAMPLER is used to obtain the energy consumption metrics. The obtained results are displayed in figure 3.9.

First of all, considering both benchmarks sets, results show an average energy saving of 26.4 %. These values ranged between the lowest energy saving percentage of 14.58 % to a peak of 45.05 %.

It is observable that synthetic benchmarks present better energy-savings compared to dependencies benchmarks. More precisely, synthetic benchmarks show an average energy saving of 33.77 % which is significantly higher than the 22.95 % average energy saving obtained by dependencies benchmarks.

This difference between the energy savings obtained by both benchmarks set is deeply connected with the V_{min} differences also obtained for both benchmark sets. In fact, the energy consumption per unit of time i.e., power, is proportional to the voltage raised to the power of two as translated in equation 3.12. Thus, a small variation in the supply voltage of a GPGPU device can greatly increase the device's energy efficiency.

$$P \propto f \times V^2 \tag{3.12}$$

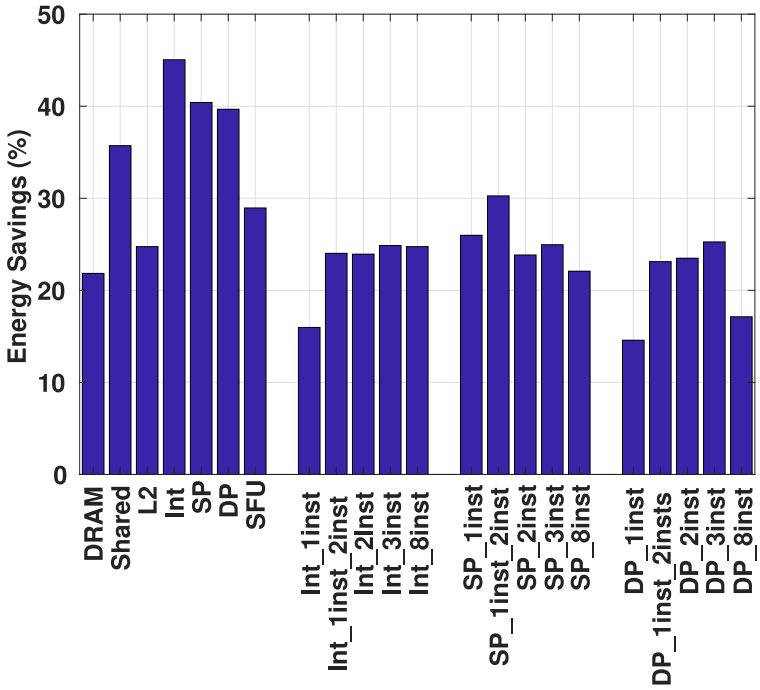


Figure 3.9: Energy savings attained by operating at V_{min} voltage on both synthetic benchmarks and benchmarks with dependencies.

3.4 Summary

A set of benchmarks each designed to stimulate a specific area of a GPGPU architecture were executed under a progressively lower supply voltage. This allowed extracting V_{min} for each benchmark, i.e. the lowest GPGPU supply voltage that still guarantees the execution correctness.

Results showed different V_{min} values across the benchmarks with variability of $0.06\,\mathrm{V}$. To further understand the observed V_{min} variability, the study was repeated to evaluate other variables, such as the device's operating frequency, temperature, aging, process variation, and voltage noise. Each studied variable made contributions to the variability, but conclusions shown voltage noise was the most preponderant.

Finally, when operating with the minimum voltage guardband, results show an average energy saving of $26.4\,\%$, ranging from the lowest energy saving percentage of $14.58\,\%$ to a peak of $45.05\,\%$.

4

Voltage guardband in deep learning applications

Contents

The focus in previous chapters was to analyze the impact of a progressively reduced voltage supply at the AMD Vega Frontier Edition GPGPU device's terminals. Results shown an energy-saving potential ranging from $14.58\,\%$ to $45.05\,\%$ with an average of $26.4\,\%$. Furthermore, conclusions demonstrated that V_{min}, and consequently the energy efficiency potential, was related to the application itself rather than other factors such as the device's operating frequency, temperature, age, or process variation.

In this manner, the goal of this chapter is to transpose the previous methodology into deep learning applications and evaluate the energy-saving potential against the precision loss that the voltage reduction process might impose.

4.1 Data Acquisition

4.1.1 V_{min} and V_{crash}

Analogously to Chapter 3, voltage guardband is the difference between nominal voltage and V_{min}, the lowest voltage at the GPGPU terminals that allows an application to run correctly. Further reducing the supply voltage below V_{min} will lead to incorrect results or the device's failure.

All deep learning models considered below produce an inference accuracy on a given data set. The model's accuracy at the nominal voltage is used as the accuracy reference, assuming the manufacturer's recommended settings will produce the correct result. Each deep learning model execution is considered correct if its accuracy is no more than $0.1\,\%$ deviated from the reference. Conversely, an accuracy deviation greater than $0.1\,\%$ is considered as an accuracy loss.

The voltage can be reduced even further allowing errors to happen, and consequently affecting the execution correctness. Naturally, this extra reduction is finite leading to the definition of V_{crash}: the voltage value at which the application is no longer executed. There are multiple types of error observed when the device's voltage is reduced below V_{min}, such as SDC, run-time errors system crashes, and indefinitely long executions.

Data corruption occurs when the execution finishes and no warning or error message is triggered, but the final result is not correct [32]. Run-time errors, which are logged by the system, occur when the program execution fails during run-time due to memory access faults. System crashes and indefinitely long executions require a manual system reboot so that the normal GPGPU device access can be restored.

Deep learning models are executed three times at all voltage levels ranging from the nominal voltage and $30\,\%$ below the nominal voltage, with a resolution of $6.25\,\mathrm{mV}$. Due to the mentioned potential errors which are more prominent below V_{min}, each execution is done after a system reboot. Rebooting the system before each program execution largely increases the experimental period, but ensures that the GPGPU is initialized at the same state across all program executions.

4.1.2 Deep Learning Framework

Implementing deep learning models requires a framework to ease and speed up models' deployment. The current study's models are executed using Tensorflow version 1.9 rc-0, as it features many open-source models and has an active community.

4.1.3 Deep learning models

Studying the impacts of the voltage reduction on deep learning applications imposes the usage of a broad set of deep learning models. This is because each model has its own architecture and consequently different management and handling of the processing unit. Thus, a wider range of models allows obtaining statistically more relevant conclusions.

Furthermore, deep learning models are organized into classes according to their architecture. Therefore, this study features deep learning models from the two main deep learning classes: CNN and RNN. Thus, capturing results from both ends of the spectrum.

The models featured in this study are all pre-trained, and the V_{min} study is conducted during the inference execution.

4.1.3.A Convolutional Neural Networks

CNNs are the state of the art on computer vision applications, including image recognition and video analysis. These models get the most attention from the community, considering the skewed amount of work featuring them compared to other neural network architectures. Due to their popularity and consequently availability, this study itself has a slight tilt towards the CNN architecture by featuring two more models from this class.

The accuracy of the models depends, not only on the model itself but also on the data they are evaluating. Thus, all the CNN models featured in this study are image classifiers and are evaluated against the same standard using the ImageNet database. ImageNet is a public image repository with over 14 000 000 images organized into more than 21 000 groups. The goal of such networks is to compute correctly each image group based on the image itself. The database owners have been running yearly competitions where participants compete with their algorithms to achieve the best image recognition accuracy. This contest has led to the popularity of some CNN featured in this study [47].

All CNN models mentioned below were obtained from TensorFlow-Slim, which an image classification model library provided by TensorFlow [48].

A – AlexNet AlexNet is one of the CNNs that became popular during the with ImageNet Large Scale Visual Recognition Challenge (ILSVRC). It achieved an accuracy that surpassed considerably the pre-

vious state of the art through the usage of deep learning, hence its recognition. Its architecture is composed of 5 convolutional layers followed by 3 fully connected layers. Some of the convolutional layers are followed by a max-pooling layer summing up to 60 million parameters and 650 000 neurons [49].

B – VGG-16 VGG-16 network is named after the VGG group from the University of Oxford, who also participated in the ILSVRC. They introduced improvements over the AlexNet network and surpassed its accuracy.

The large kernels in the first layers of the AlexNet architecture were replaced by smaller ones and consecutive ones. This modification further increases the depth of these models. This increasing depth with lower kernel sizes is known to enable the recognition of more complex features from the input images, thus improve its performance.

As expected, the improvements come at the cost of higher computational and memory requirements, due to the increased number of parameters.

C – VGG-19 VGG-19, as the name suggests, is similar to VGG-16. They are the best performing models from the VGG group. The difference lies in the number o layers each one has. Whilst VGG-16 has 16 weight layers, VGG-19 has 19 [50].

D – Inception The Inception network, which is also called GoogLeNet, is also a convolutional neural network and was the winner of ILSVRC in the year 2014. It pushed the top 5 test error further to $6.6\,\%$.

Google introduced the inception module with this network. Instead of stacking convolutional layers, an inception module is an approach where different sized kernel convolutions, namely a 1×1, a 3×3, and a 5×5, are executed in parallel at the same level as shown in Figure 4.1.

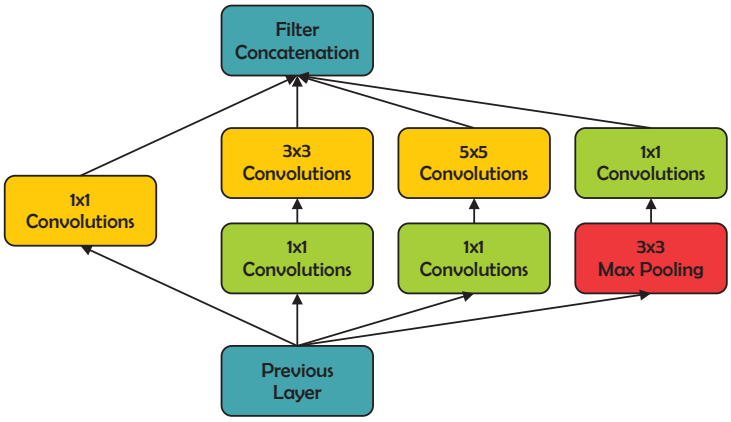


Figure 4.1: Diagram of the inception module.

Having different sized kernels executed in parallel at the same level revealed to be advantageous due to the unpredictability of the input image's scale. The input's detail, that the network is trying to

classify, can either occupy a small or a big portion of the whole image. However, large kernels thrive when the detail is large, whilst small kernels thrive when the detail is smaller. The inception module has both large and small kernels whose result is passed into the following module, hence its success.

The used inception network (InceptionV4) is composed of a total of 9 inception modules, each of these having a depth of 2 layers as shown in Figure 4.1. In total the network has 22 layers and close to 6.8 million parameters. There are no fully connected layers that substantially reduces the number of parameters when compared to the AlexNet and VGG networks. This property of the inception network also allows it to execute faster when compared to the mentioned networks [51, 52].

E – ResNet Empirical knowledge shows that increasing the depth of a neural network should also increase its accuracy. However, some issues arise with this approach, namely over-fitting, the vanishing gradients, and the degradation of the training accuracy.

Considering two neural networks: one whose architecture matches the first but with a few added layers. Since the added layers on the deeper network can just preserve the result of the shallower architecture, it is expected for the first network to produce the same or less training error than the latter. However, experiments reveal that deeper networks are unable to attain equally good solutions within a feasible time. This phenomenon is called the degradation of training accuracy. In short, it states that the optimization of different architectures does not require the same effort.

Microsoft introduced the residual neural network, ResNet, along with a new module called a residual block that tackles the degradation issue. It does so by creating a direct path between the input and the output of the module, i.e. the input of the module is added to the output as shown in the figure 4.2. Instead of the layers optimize a given mapping, the module forces them to optimize the residual mapping instead.

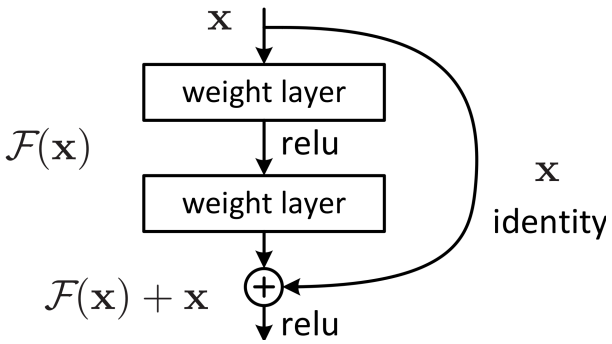


Figure 4.2: Diagram of the residual module [53].

This module allowed the network to have a total of 152 layers: 8 times more layers than the VGG network. Furthermore, ResNet was the first network to surpass human accuracy on the ImageNet database classification, which granted this network first place on ILSVRC 2015 on the classification

task [53].

F – Inception-ResNet Based on the success of the ResNet, Szegedy et al. [52] created hybrid architectures of the inception network using residual modules. These models were able to achieve better accuracies when compared to their counterparts.

This study features Inception-ResNet-v2, which is the top performer of the hybrid inception networks. This network's computational cost is similar to InceptionV4, also featured in this study.

4.1.3.B Recurrent Neural Networks

On the other side of the spectrum, there are RNNs which became popular due to their capabilities in processing sequential data, including text and voice.

These models are known to have limited data parallelism and high data dependencies when compared to CNNs. This limitation is caused by the complex input data these models process, which leads to inefficient resource utilization [54].

Considering these limitations and recalling the slight worse V_{min} results that dependency benchmarks obtained in the previous chapter 3, it is expected for these models to also perform slightly worse when compared to CNN.

A – Skip-Thoughts Skip-thoughts is a sentence encoder that predicts the surroundings of an input sentence. It uses an encoder-decoder model: On one hand, the encoder is responsible for mapping each sentence into a vector, called skip-thought vectors. Naturally, the encoder will generate similar vectors when the input sentences share semantic and syntactic properties. On the other hand, the decoder is used to predict the sentences, based on the vector provided by the encoder. Both the encoder and the decoder are neural networks featuring an RNN architecture [55].

This study features a trained version of the Skip-Thoughts neural network trained under the BookCorpus dataset, which is a collection of novels written by unpublished authors [56]. This network uses an unsupervised learning approach since the goal is to learn generic sentence representations, i.e. representations that are not bound to a previously predetermined interpretation. This approach contrasts with supervised learning that requires each sentence of the training set to have a label class, i.e. a predetermined interpretation of the sentence.

In order to benchmark the Skip-thoughts network against other sentence representation learning methods, Kiros et al. [55] used several experiments where the encoder is used as a feature extractor, thus allowing for quantitative evaluations. One such experiment uses the question-type classification (TREC) dataset. The dataset is a set of questions organized into classes such as abbreviations, entities, descriptions, human, locations, and numeric. The experiment consists of encoding those questions

using the Skip-thoughts encoder and evaluate the vectors' proximity of those questions that belong to the same subclass, using a logistic regression classifier. The model achieved an accuracy of $92.2\,\%$ under this experiment [55].

As part of the current study, this experiment is reproduced under the V_{min} endeavor, as described in the following subsection 4.1.1. The final performance is to be measured after each execution and compared against the state of the art accuracy.

B – Sentiment Radford et al. [57], from OpenAI, published a RNN whose goal is to automatically generate new product reviews or continue existing ones. The network was trained on a corpus of Amazon product reviews collected from May 1996 to July 2014 containing over $82\,\text{million}$ reviews [58].

This model's RNN architecture is a single layer with multiplicative LSTM units, amounting to a total of 4096 units. The authors discovered that one of these units mapped directly to the review's sentiment. Meaning that a specific neuron would be activated or deactivated depending on the reviews' tone. Furthermore, explicitly changing the activation of this unit forces the network to output positive or negative reviews.

The trained version of the model is used in inference mode under a similar task to the Skip-thought network, to evaluate its accuracy. Given that this model also produces its language representation, i.e. the unit's activation values, it is possible to evaluate how close are the representations of pairs of sentences extracted from Microsoft Research Paraphrase Corpus (MRPC). MRPC is a dataset containing pairs of sentences that are labeled by humans identifying the pair semantic relatedness [59]. This procedure is used under the V_{min} study.

C – ReactionRNN ReactionRNN is a RNN that can predict proportionate reactions from a 140 character input. The set output set of reactions include *love*, *wow*, *haha*, *sad*, *angry*. The text is handled at the character level, contrasting with the Skip-thought network that handles the input at the sentence level.

The model's architecture is composed of two layers. The first layer contains 256 Gated Recurrent Unit (GRU) that is followed by a fully-connected layer with 5 output nodes. Each output node corresponds to one of the output classes. The model is trained on a collection of Facebook statuses' reactions [60].

Analogously to the Sentiment network, the current V_{min} study will use MRPC to evaluate the model's accuracy. The closeness of the sentence pairs from the data set translates into the model's accuracy [59].

4.1.3.C Other Neural Networks

RNNs are state of the art regarding Natural Language Processing (NLP) applications, including language modeling and machine translation. Consequently, literature pushing these implementations forward is largely available. However, the architecture itself inherently processes information sequentially, which imposes limitations to data parallelization.

There are, however, different architectures with state of the art performance on NLP challenges. This study features one such architecture.

A – BERT Bidirectional Encoder Representations from Transformers (BERT) is a NLP technique to create language representations. More precisely, it produces a contextual representation of a word. Contextual, in this scenario, means that the representation of a given word will vary depending on the other words surrounding it within the same sentence. Furthermore, contextual models can either be unidirectional or bidirectional depending if the word is contextualized using only words from one of its sides, e.g. contextualized only using words to its left, or from both sides, respectively. BERT, however, produces bidirectional context language representations that are used by Google to enhance their understanding of Google searches [61, 62].

Vaswani et al. [63] introduced the Transformer architecture intending to remove the sequential dependencies, by relying exclusively on their own version of an attention mechanism. The attention mechanism is an attempt to improve RNNs capability of relating words that are further apart. Originally, the module was a simple feed-forward network with one hidden layer. It works by creating weighted shortcuts from the input hidden state to the decoder, thus allowing all encoder's hidden state to potentially have a direct impact on the decoder.

BERT relies on the Transformer architecture, making it a multi-layer bidirectional Transformer encoder. Analogously to the Skip-though network, BERT was trained under the Book-Corpus dataset plus a data collection obtained from Wikipedia.

The current study uses the trained version of the base implementation of BERT obtained from Google's research repository [64]. This version is used in inference mode in the same task as the Sentiment network. The MRPC dataset, containing pairs of sentences that are labeled by humans identifying the pair semantic relatedness, is used to evaluate BERT's accuracy within the V_{min} study [59].

4.2 Classification accuracy

This study features a heterogeneous set of deep learning applications. Each one has its own architecture and purpose. Consequently, it is not possible to evaluate all models under the same task, but having

such a set up is also not in the scope of this study. The focus is to evaluate the degree to which the accuracy is impacted by varying the GPGPU's supply voltage.

Nonetheless, there was an effort to standardize the accuracy extraction method. All CNN models, focused on image recognition, are evaluated based on their performance on the ImageNet dataset. Whilst RNN models and BERT, focused on NLP, are evaluated based on their performance on MRPC dataset.

Table 4.1 depicts the obtained accuracy for all the deep learning models, at their respective tasks, with the default nominal voltage at the AMD Vega Frontier Edition GPGPU device's terminals.

Table 4.1: Deep learning models inference accuracy at nominal voltage.

DEEP LEARNING MODEL	ACCURACY
AlexNet	83.0 %
VGG-16	89.8 %
VGG-19	89.8 %
Inception V4	95.2 %
ResNet V2	94.1 %
Inception-ResNet	95.3 %
Skip-Thoughts	92.2 %
Sentiment	73.0 %
ReactionRNN	61.3 %
BERT	89.3 %

Based on the benchmark's results from the chapter 3, it is known that the application itself is the main responsible for the V_{min} variation. It is also known that GPGPU's memory management has a greater sensitivity to voltage reduction.

With this in mind, and bearing that deep learning applications are highly demanding of computing resources, it is expected for V_{min} to be substantially higher than on the previous benchmarks. Consequently, deep learning models should also experience a lower range of energy-saving potential.

Furthermore, results on chapter 3 did not display any SDC errors, i.e. all benchmarks either performed correctly or didn't perform at all at each voltage level. This was explained based on the application's lower complexity. However, SDC is expected to occur for deep learning models, allowing for precision to energy efficiency trade-off.

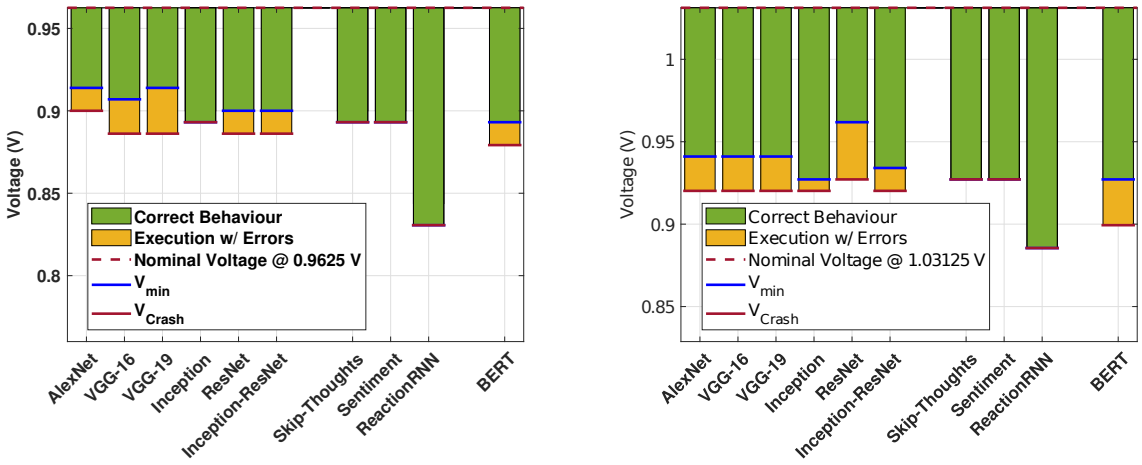
4.3 Voltage Guardband Accuracy Impact

4.3.1 Minimum operating voltage

As it was previously stated V_{min} is defined as the minimum voltage that ensures correct execution. For the deep learning case, it corresponds to the minimum voltage that ensures a result within an error range

of 0.1 % regarding the reference accuracy. Conversely, V_{crash} is the voltage level where the accuracy loss might be greater than 0.1 %, but the execution is still terminated successfully.

Figure 4.3 depicts the V_{min} and V_{crash} absolute values obtained at a lower and higher GPGPU core frequency set up, 1028.57 MHz, on the figure 4.3(a), and 1107.69 MHz, on the figure 4.3(b), respectively.



(a) 1028.57 MHz

(b) 1107.69 MHz

Figure 4.3: Obtained V_{min} and V_{crash} for deep learning models.

First of all, empirically, all deep learning models excluding ReactionRNN show a similar accuracy sensitivity to the voltage guardband reduction. The average V_{min} is 0.900 06 V and 0.941 25 V on the lower and higher frequency experiment respectively.

Normalizing V_{min} results from both frequencies based on the nominal value, further confirms the obtained the results on subsection 3.3.2.A. Lower frequencies achieve higher V_{min} readings, thus resulting in a lower voltage guardband. At 1028.57 MHz, the nominal voltage set by the GPGPU manufacturer is 0.962 50 V resulting in an average normalized V_{min} of 0.935 71. Conversely, at 1107.69 MHz, the nominal voltage set by the GPGPU manufacturer is 1.031 25 V resulting in an average normalized V_{min} of 0.912 73. This corresponds to an undervoltage of 6.43 % and 8.73 % when compared to the nominal voltage of each frequency, 0.9625 V and 1.031 25 V, respectively.

The standard deviation of the V_{min} readings is 0.021 V and 0.017 V for the lower and higher GPGPU core's frequency respectively. By excluding ReactionRNN results, the standard deviation calculation decreases significantly to 0.008 V and 0.010 V respectively. This is caused by the smaller architecture of ReactionRNN that features substantially fewer parameters than its counterparts.

It is interesting to note that none of the RNN models showed an accuracy loss with the voltage supply reduction. All successful executions of these models did not incur in accuracy loss, meaning $V_{crash} = V_{min}$.

On the other hand, CNN models along with BERT displayed SDC occurrences resulting in an accuracy loss, hence $V_{crash} < V_{min}$. These models allowed further decreasing of the supply voltage by an average of 0.005 09 V and 0.013 13 V on each frequency experiment, respectively. Therefore, there is a lower precision opportunity within this supply voltage range.

In sum, the results at 1028.57 MHz reveal that there is a voltage guardband, that can be safely reduced on deep learning applications, ranging from 4.55 % up to 12.33 % with an average of 6.43 %. The voltage guardband can be further reduced, up to 2.60 %, thus allowing accuracy loss. Likewise, at 1107.69 MHz the voltage guardband can be safely reduced from 6.06 % up to 12.73 % with an average of 8.75 %. By allowing accuracy loss, the voltage guardband can be further decreased up to 3.03 % on some of the tested models.

4.3.2 Accuracy loss

The previous subsection revealed there is a lower precision energy efficiency opportunity by reducing the GPGPU supply voltage further below V_{min}. This opportunity gap corresponds to a maximum of 2.60 % and 3.03 % undervoltage, when the device's core frequency is set to 1028.57 MHz and 1107.69 MHz respectively. The purpose of this subsection is to evaluate the accuracy behavior within this undervoltage range.

Table 4.2 depicts the obtained accuracy at V_{crash} for all deep learning models, at their respective tasks, at both studied operating frequencies. The table also includes, within brackets, the accuracy loss when compared to the nominal accuracy whose values are depicted in the table 4.1 above.

Table 4.2: Deep learning models inference accuracy at V_{crash} at 1028.57 MHz and 1107.69 MHz, including the accuracy loss when compared to the reference accuracy.

DEEP LEARNING MODEL	REFERENCE ACCURACY	1028.57 MHz	1107.69 MHz
AlexNet	83.0 %	81.68 % (1.32 %)	76.58 % (6.42 %)
VGG-16	89.8 %	38.57 % (51.23 %)	76.24 % (13.56 %)
VGG-19	89.8 %	28.28 % (61.52 %)	65.31 % (24.49 %)
Inception V4	95.2 %	95.12 % (0.08 %)	95.07 % (0.13 %)
ResNet V2	94.1 %	89.14 % (4.96 %)	86.77 % (7.33 %)
Inception-ResNet	95.3 %	93.18 % (2.12 %)	94.44 % (0.86 %)
Skip-Thoughts	92.2 %	92.19 % (0.01 %)	92.19 % (0.01 %)
Sentiment	73.0 %	72.99 % (0.01 %)	73.00 % (0.00 %)
ReactionRNN	61.3 %	61.30 % (0.00 %)	61.30 % (0.00 %)
BERT	89.3 %	84.61 % (4.69 %)	55.95 % (33.35 %)

Results, when operating at the low frequency, 1028.57 MHz, show accuracy droops up to 61.52 % with an average of 12.59 % when working at V_{crash} supply voltage. Conversely, when operating at the high frequency, 1107.69 MHz, the accuracy droop achieved only a maximum of 33.35 % with an average of

$8.62\,\%$.

It is also important to note that there are models that did not experience any accuracy loss at all (Skip-Thoughts, Sentiment, and ReactionRNN) and a few more that had only a residual impact on the accuracy when working near failure supply voltages (InceptionV4, and Inception-ResNet).

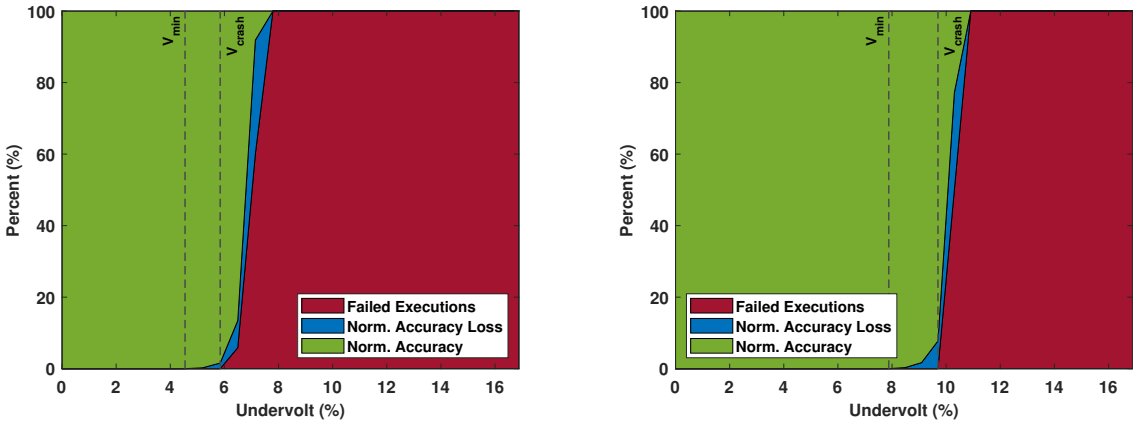
The remaining of the current subsection displays the inference accuracy distribution across an increasing undervolt percentage.

4.3.2.A Convolutional Neural Networks

According to the previous subsection, CNN models displayed a gap between V_{min} a V_{crash}. This means there is a voltage band where all executions are completed successfully but there is at least a $0.1\,\%$ accuracy loss.

To further characterize this issue, the execution of each CNN model is displayed in the figures 4.4 to 4.9. In these graphs, the y axis presents the percentage of failed executions out of a total of 10. While the x axis presents the undervolt percentage (i.e., regarding the nominal voltage at the corresponding frequency). The green zone corresponds to a condition where all executions presented an error lower than $0.1\,\%$. The blue zone corresponds to a case of complete execution, but with accuracy losses; and the red zone represent cases of incomplete/failed executions.

Furthermore, each graph has two vertical dashed lines corresponding to V_{min} and V_{crash}. Naturally, the dashed line further to the left represents V_{min} whilst the dashed line further to the right represents V_{crash}. However, these lines might overlap meaning $V_{min} = V_{crash}$, i.e. there was no accuracy loss for the respective deep learning model.

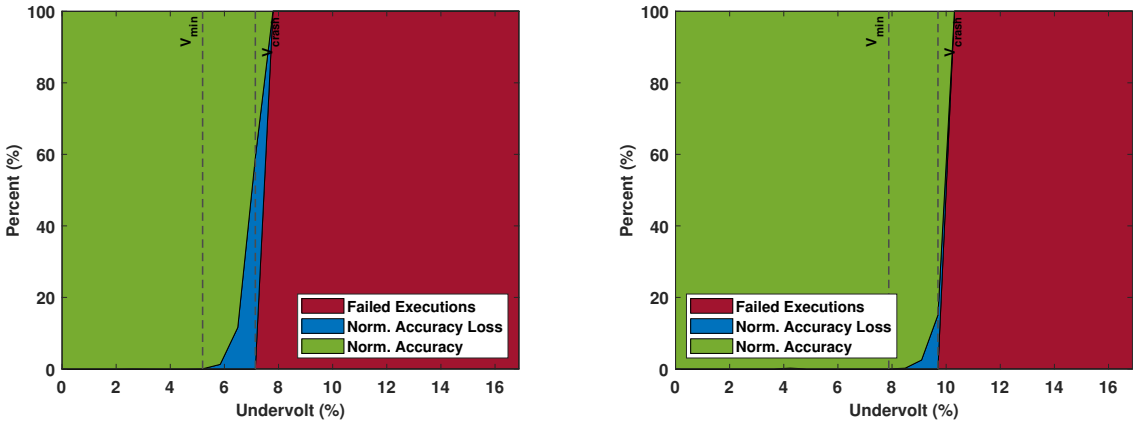


(a) AlexNet @ 1028.57 MHz

(b) AlexNet @ 1107.69 MHz

Figure 4.4: AlexNet inference accuracy distribution across an increasing undervolt percentage.

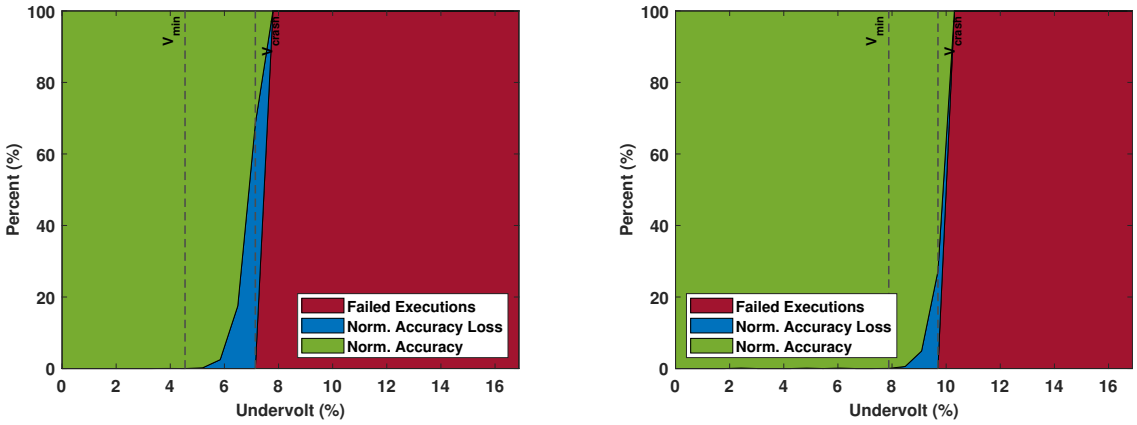
First of all, it is important to note that the undervoltage increments have a resolution of $6.25\,\mathrm{mV}$. This



(a) VGG-16 @ 1028.57 MHz

(b) VGG-16 @ 1107.69 MHz

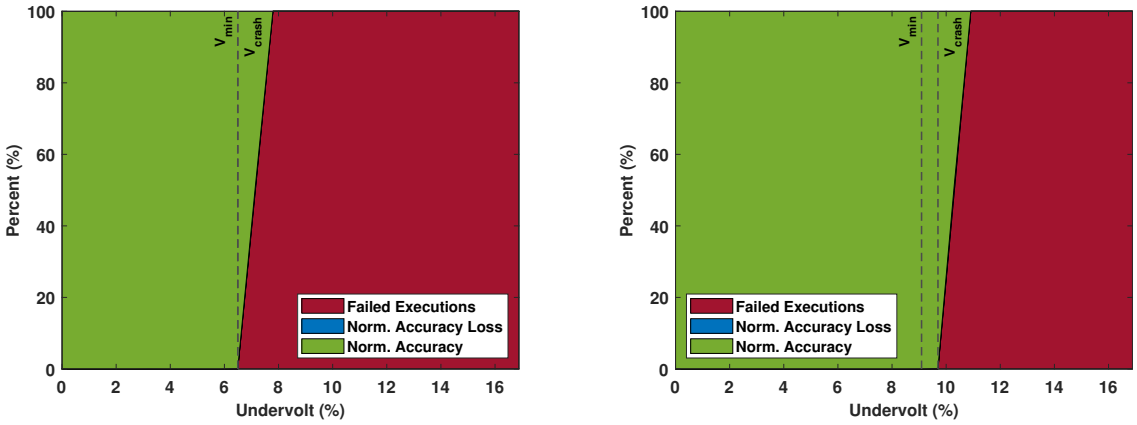
Figure 4.5: VGG-16 inference accuracy distribution across an increasing undervolt percentage.



(a) VGG-19 @ 1028.57 MHz

(b) VGG-19 @ 1107.69 MHz

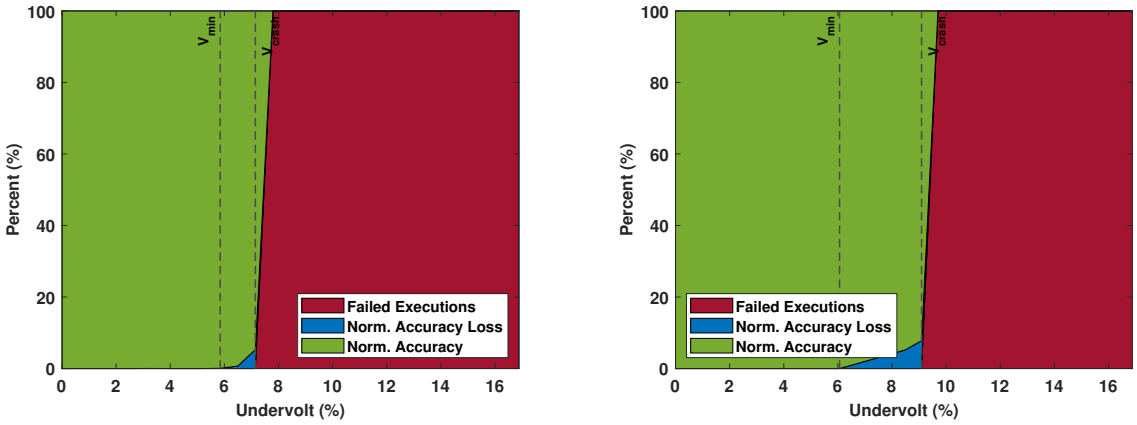
Figure 4.6: VGG-19 inference accuracy distribution across an increasing undervolt percentage.



(a) Inception @ 1028.57 MHz

(b) Inception @ 1107.69 MHz

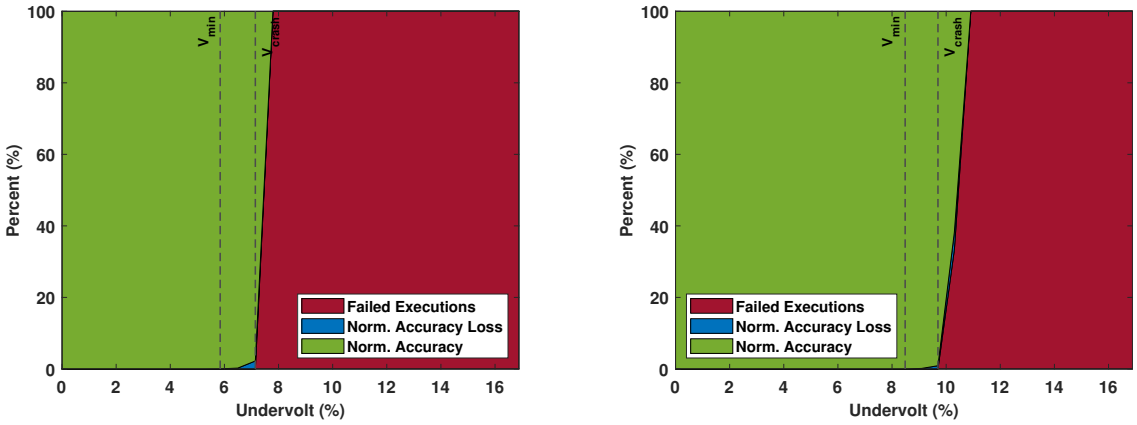
Figure 4.7: Inception inference accuracy distribution across an increasing undervolt percentage.



(a) ResNet @ 1028.57 MHz

(b) ResNet @ 1107.69 MHz

Figure 4.8: ResNet inference accuracy distribution across an increasing undervolt percentage.



(a) Inception-ResNet @ 1028.57 MHz

(b) Inception-ResNet @ 1107.69 MHz

Figure 4.9: Inception-ResNet inference accuracy distribution across an increasing undervolt percentage.

value is bounded by the AMD Vega Frontier Edition GPGPU device's tools. Therefore, the resolution corresponds to an undervoltage of $0.65\,\%$ and $0.61\,\%$ when the device's cores are working at 1028.57 MHz and 1107.69 MHz respectively.

Empirically, one can verify from the figures 4.4 to 4.9 that decreasing the GPGPU device's supply voltage further below V_{crash} undervoltage, still produces successful executions. Unfortunately, some of those executions fail due to run-time errors, system crashes, or indefinitely long executions.

The models with the most data corruption impact at V_{crash} are VGG-16 and VGG-19. In both frequency experiments, these models have the greatest amount of accuracy loss. The VGG models are characterized by their high computationally requirements. These are the models with the most naively stacked convolutions layers featured in this study.

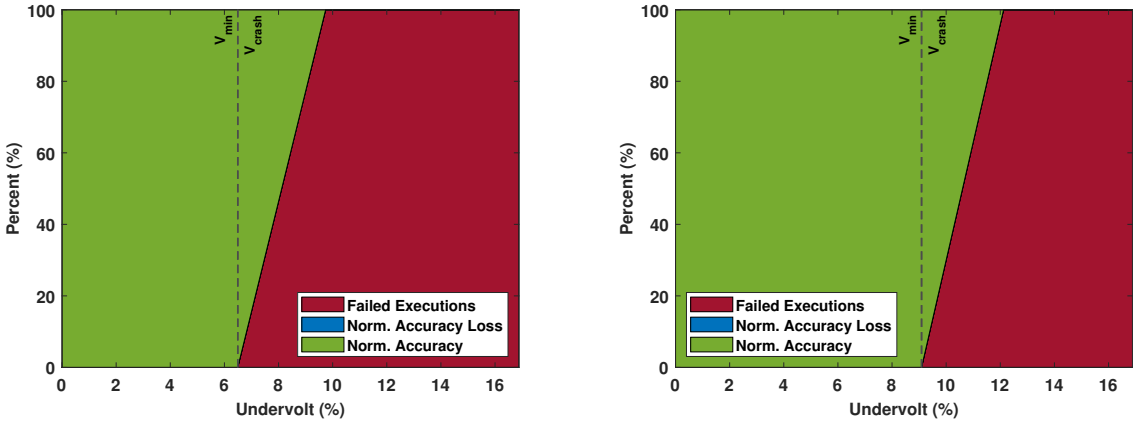
On the other end of the spectrum, Inception has no data corruption on the lower frequency experiment, having $V_{min} = V_{crash}$. On the higher frequency experiment, the gap between V_{min} and V_{crash} is the bare minimum, matching the GPGPU device's tool resolution of 6.25 mV. The accuracy loss is $0.16\,\%$ which is barely noticeable on the graph.

There is also an apparent tendency for lower frequencies to display higher data corruption rates. VGG-16, VGG-19, Inception, and ResNet all have a higher precision loss on their lower frequency experiment.

4.3.2.B Recurrent Neural Networks

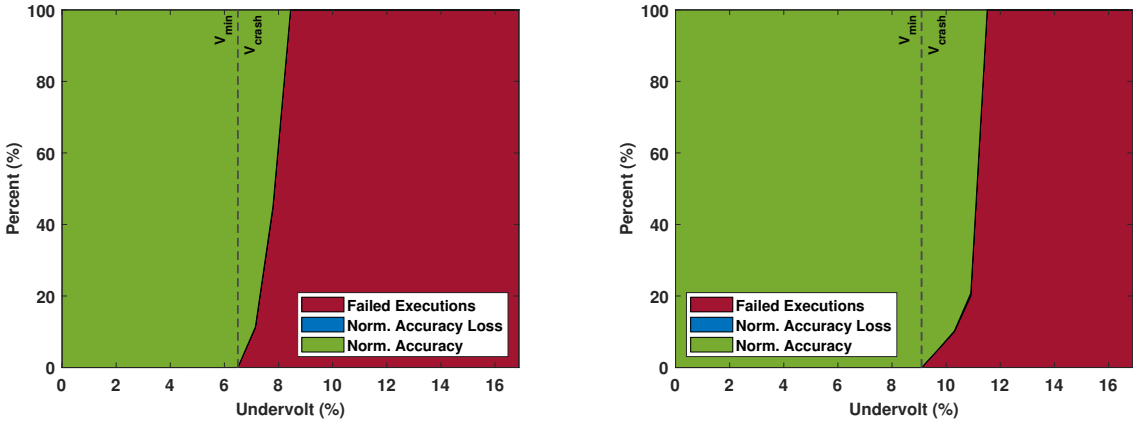
Figures 4.10 to 4.12 depict the accuracy loss evolution with an increasing undervoltage percentage for RNN models. As inferred from the figure 4.3 in the previous subsection, none of the RNN models had V_{crash} results different than V_{min}. Meaning that all models were either executed successfully without

data corruptions or at least one of the executions failed.



(a) Skip-Thoughts @ 1028.57 MHz **(b)** Skip-Thoughts @ 1107.69 MHz

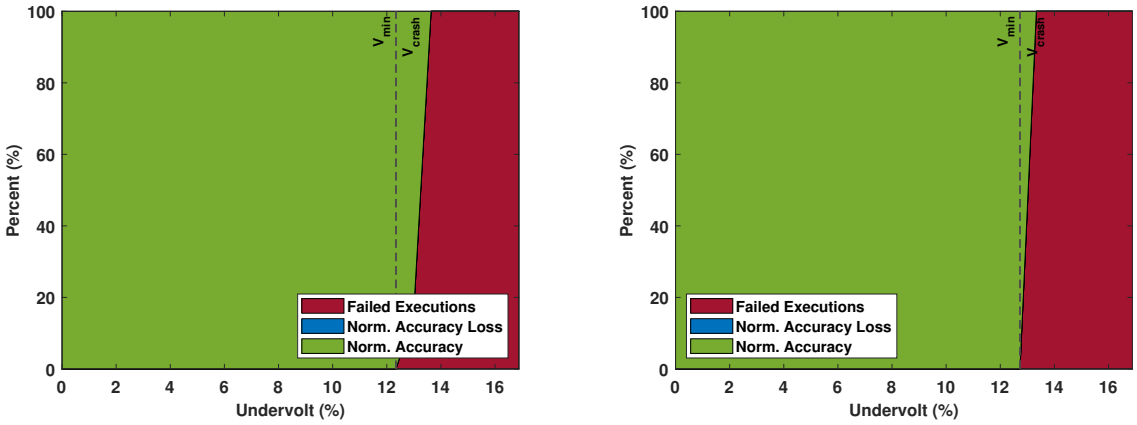
Figure 4.10: Skip-Thoughts inference accuracy distribution across an increasing undervolt percentage.



(a) Sentiment @ 1028.57 MHz **(b)** Sentiment @ 1107.69 MHz

Figure 4.11: Sentiment inference accuracy distribution across an increasing undervolt percentage.

None of the RNN models had SDC occurrences even when the GPGPU device's supply voltage was below V_{crash}. Also, each model has a different evolution towards failed execution. Skip-thoughts had a less steep progression, meaning that progressively more and more executions were failing with the increased undervoltage. This contrasts with Sentiment and ReactionRNN that had an abrupt change in that regard.

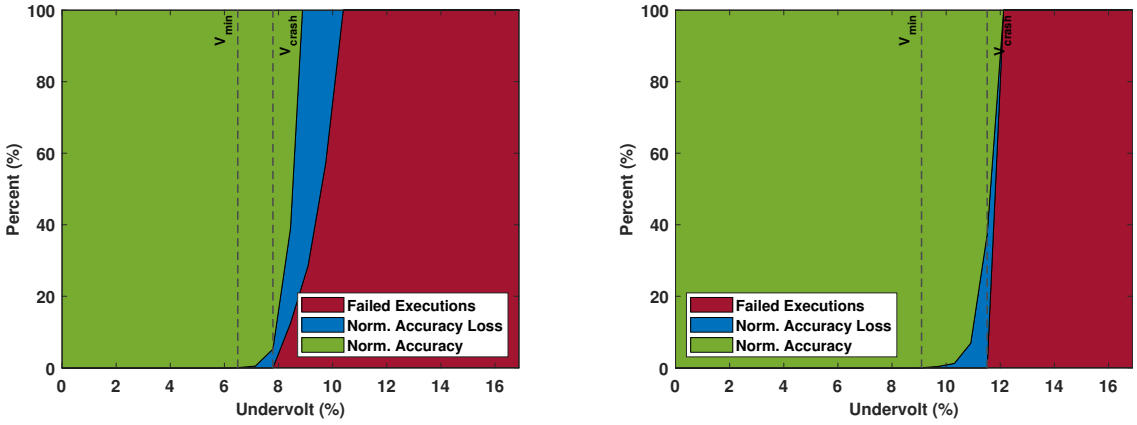


(a) ReactionRNN @ 1028.57 MHz **(b)** ReactionRNN @ 1107.69 MHz

Figure 4.12: ReactionRNN inference accuracy distribution across an increasing undervolt percentage.

4.3.2.C Other Neural Networks

Finally, the figure 4.13 depicts BERT's accuracy loss progression with an increasing undervoltage percentage.



(a) BERT @ 1028.57 MHz **(b)** BERT @ 1107.69 MHz

Figure 4.13: BERT inference accuracy distribution across an increasing undervolt percentage.

Unlike the CNN models, BERT has higher data corruption readings on a higher frequency. It is also the only model to have drastically different progression patterns on both frequencies studied.

On one hand, the low-frequency experiment had most of the data corruption incidents occurring below V_{crash} with the amount of failed executions progressively increasing after that point. On the other hand, the high-frequency experiment had considerably more data corruption at V_{crash} and the amount of failed executions increased abruptly after that point. The accuracy loss at V_{crash} with 1028.57 MHz is

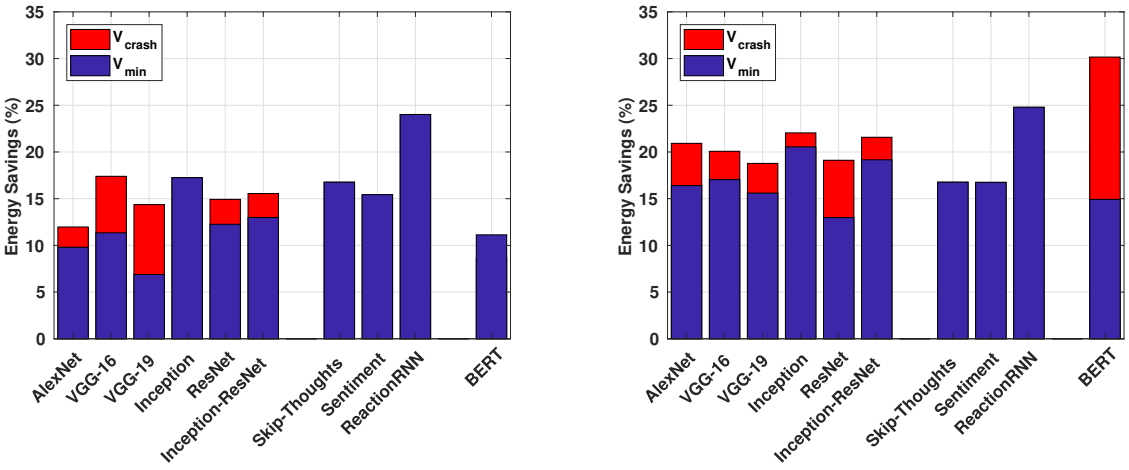
0.41 % compared to a 33.35 % when the device's core frequency is 1107.69 MHz.

4.4 Voltage Guardband Energy Impact

Having V_{min} and V_{crash} results for each deep learning model along with the average power consumption metric collected by gpowerSAMPLER, it is now possible to evaluate the energy efficiency attained with the current study. Knowing the average power consumption and also execution time we can infer the energy consumption for each execution through the equation 4.1.

$$E = \frac{P}{\Delta t} \tag{4.1}$$

Figure 4.14 depicts the obtained energy saving results for the deep learning models featured in the current chapter. Results on low frequency experiment, guaranteeing the execution correctness, show an energy-saving potential ranging from 6.88 % up to 24.01 % with an average of 13.79 %. The average energy-saving potential can be further increased by an average of 1.84 % by allowing the model's accuracy to drop.



(a) 1028.57 MHz **(b)** 1107.69 MHz

Figure 4.14: Energy savings attained by operating at V_{min} and V_{crash} voltage on deep learning models.

Analogously, on the high frequency experiment, the energy saving potential ranges from 12.97 % up to 24.79 % with an average of 17.50 %. This potential can be further increased from 3.77 % to 5.38 % with an average of 3.60 % by allowing the model's accuracy to drop.

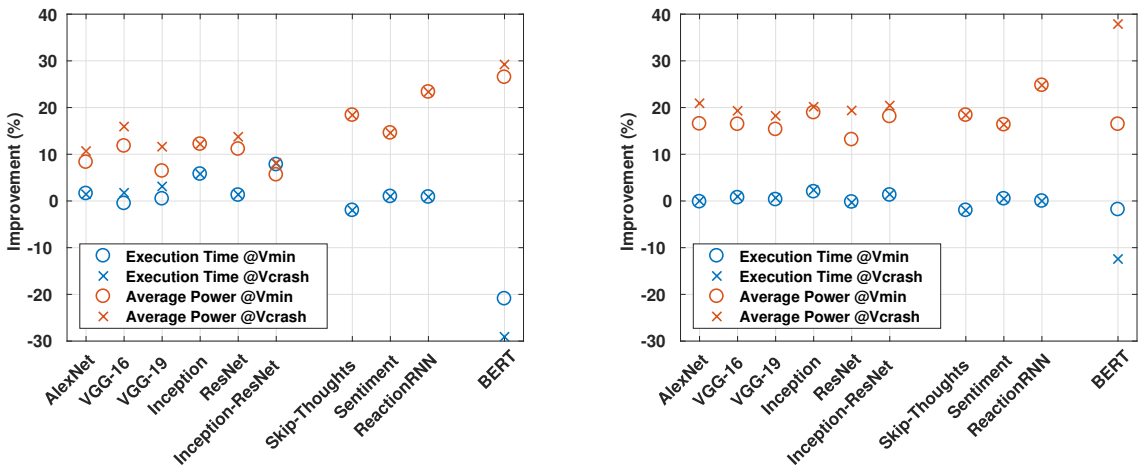
Naturally, given V_{min} results, ReactionRNN achieved the highest energy consumption improvement at V_{min}. This improvement is only surpassed at the high frequency by the BERT model when the supply

voltage is at V_{crash}.

By splitting the energy saving results across the deep learning models architectures: CNN models alone achieved an energy consumption improvement ranging from 12.97 % up to 20.55 % with an average of 17.50 %. Conversely, RNN models achieved an energy consumption improvement ranging from 16.75 % up to 24.79 % with an average of 19.43 %. While these results are retrieved from the high-frequency experiment, similar conclusions can be inferred from the lower frequency experiment.

Also, CNN models achieved an average energy consumption improvement slightly below RNN. However, it is notable that it did so with approximately half the standard deviation error. This metric is 2.68 % for the CNN architecture and 4.64 % for the RNN architecture.

As mentioned above, two metrics have an impact on the overall energy consumption: execution time and average power consumption. To further comprehend how each of these metrics had an impact on the results shown in the figure 4.14, the graphs in figure 4.15 are presented. These graphs portray the improvement from both execution time and average power consumption metrics compared to the reference execution.



(a) 1028.57 MHz

(b) 1107.69 MHz

Figure 4.15: Execution time and average power consumption improvements at V_{min} and V_{crash} over the same metrics at the nominal voltage.

Results also confirm the initial expectation that the average power consumption decreases with the undervolting process, given the known relationship between power (P) and the supply voltage (V) represented in the equation 4.2,

$$P \propto f \times V^2 \tag{4.2}$$

where f represents the GPGPU device's operating frequency. In particular, the power consumption has an average improvement of $13.83\,\%$ and $17.44\,\%$ on the low and high frequency experiment, respectively. These averages match the energy savings improvements on both frequencies, meaning the GPGPU device's power consumption is the driver of such energy-saving improvements.

Additionally, the execution time does remain stable at a lower supply voltage compared to the reference execution at the nominal voltage. As an illustrative example, the average execution time improvement on the high-frequency experiment is $0.06\,\%$ with a standard deviation of $1.24\,\%$. There were executions slightly faster than the reference and executions slightly slower than the reference, translating into close to no impact on the energy savings.

However, the BERT results on the low-frequency experiment are worthy of notice. At $1028.57\,\text{MHz}$, the execution speed drastically decreased thus trumping the improvements obtained from the lower power required. At V_{crash} levels, the execution speed dropped even lower making the V_{crash} energy-consumption higher than the result at V_{min}. This explains why the BERT model did obtain a V_{crash} lower than V_{min} during inference, but there were no energy-saving improvements displayed in figure 4.14.

Overall, voltage guardband exploitation provides relevant improvements in the context of energy efficiency maximization for deep learning applications.

4.5 Summary

The current chapter evaluates how deep learning models, including CNNs and RNNs, perform when working at near-failure supply voltages.

Results showed an undervolt potential up to $9\,\%$ whilst guaranteeing the reference accuracy on the studied models. With that setup, results also showed an energy savings potential up to $24.79\,\%$ with an average of $15.35\,\%$.

Further decreasing the GPGPU device's supply voltage leads to accuracy droops up to $61.52\,\%$ with an average of $12.59\,\%$, but further increases the energy savings potential up to $30.16\,\%$ with an average of $18.37\,\%$.

5

Conclusions

Contents

The present thesis proposes an approach to study the energy savings potential of modern deep learning applications on modern GPGPU devices, using a AMD Radeon Vega Frontier Edition GPGPU as a case study.

First of all, the GPGPU device's voltage guardband is characterized using benchmarks. To do so, two sets of synthetic benchmarks were used. The first includes applications that target a specific GPGPU architecture, such as the ALU unit. The latter extends the benchmark application set by introducing data dependencies.

Results show an undervoltage potential ranging from ranging from $16.9\,\%$ to $20.7\,\%$ with an average of $15.68\,\%$ on the synthetic benchmarks set, and ranging from ranging from $11.04\,\%$ to $12.34\,\%$ with an average of $11.60\,\%$ on the synthetic benchmarks set. Thus, confirming the expectations that dependency benchmarks would obtain higher V_{min} readings, i.e. a lower voltage guardband.

When operating at V_{min}, the GPGPU device achieved a energy efficiency ranging from $14.58\,\%$ up to $45.05\,\%$ with an average of $26.4\,\%$.

The benchmark results have also shown a V_{min} variability of $0.06\,\text{V}$ corresponding to $6.2\,\%$ when compared to the nominal voltage. An analysis, bearing in mind potential causes for the V_{min} variability, concluded that it is deeply connected with the application itself. The device's operating frequency, temperature, aging, process variation, and inter-kernel executions all rendered an insufficient V_{min} variability to explain the variability magnitude observed in the benchmarks.

Knowing the application itself is the root cause of the V_{min} variability, deep learning models were introduced where the same endeavor was repeated.

Results showed deep learning models can achieve energy savings of up to $24.79\,\%$ with an average of $15.35\,\%$ whilst guaranteeing the nominal accuracy. Furthermore, by working at V_{crash}, energy efficiency can be increased by an average of $2.72\,\%$ at the expense of the model's accuracy. When the GPGPU is set to work at near failure supply voltages, V_{crash}, the observed accuracy droop achieved an average of $10.61\,\%$ and a maximum of $61.52\,\%$.

Additionally, there were executions further below V_{crash} that achieved even higher energy-saving results. Obviously, given V_{crash} definition, there were also failed executions at those voltage levels, hence those executions were disregarded. Nonetheless, this means V_{crash} is not the ultimate limit on the voltage guardband reduction approach when seeking lower power consumption.

5.1 Future work

Based on the results showing tempting energy-saving opportunities, this work could be pushed forward by implementing a model that would predict an application's minimum operating voltage at a predefined maximum accuracy loss. This model could then be used to dynamically set the GPGPU supply voltage

based on the application itself, in a similar fashion current DVFS techniques are doing with the frequency scaling.

Bibliography

[1] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 620–629.

[2] S. Mittal and S. Vaishay, "A survey of techniques for optimizing deep learning on gpus," *Journal of Systems Architecture*, vol. 99, p. 101635, 2019.

[3] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, and V. J. Reddi, "Safe limits on voltage reduction efficiency in gpus: a direct measurement approach," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 294–307.

[4] J. Leng, Y. Zu, and V. J. Reddi, "Gpu voltage noise: Characterization and hierarchical smoothing of spatial and temporal voltage noise interference in gpu architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 161–173.

[5] Ian Goodfellow and Yoshua Bengio and Aaron Courville, *Deep Learning*. MIT Press, 2016.

[6] J. G. Carbonell, T. M. Mitchell, and R. S. Michalski, *Machine learning: An artificial intelligence approach*. M. Kaufmann., 1983.

[7] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell, "Decaf: A deep convolutional activation feature for generic visual recognition," in *International conference on machine learning*, 2014, pp. 647–655.

[8] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: an astounding baseline for recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2014, pp. 806–813.

[9] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving deep neural networks for lvcsr using rectified linear units and dropout," in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 8609–8613.

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[11] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[12] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS workshop*, no. CONF, 2011.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[14] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[15] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 2135–2135.

[16] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*. IEEE, 2016, pp. 99–104.

[17] NVIDIA, "GPU-Based Deep Learning Inference: A Performance and Power Analysis," NVIDIA Corporation, Tech. Rep., 2015.

[18] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[19] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with COTS HPC systems," in *International Conference on Machine Learning*, 2013, pp. 1337–1345.

[20] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," NVIDIA Corporation, Tech. Rep., 2009.

[21] AMD, "AMD Graphics Core Next (GCN) Architecture," Advanced Micro Devices, Inc., Tech. Rep., 2012.

[22] NVIDIA, “NVIDIA Tesla V100 GPU Architecture,” NVIDIA Corporation, Tech. Rep., 2017.

[23] AMD, “Radeon’s next-generation Vega architecture,” Advanced Micro Devices, Inc., Tech. Rep., 2017.

[24] ——, “Dissecting the Polaris Architecture,” Advanced Micro Devices, Inc., Tech. Rep., 2016.

[25] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, “cudnn: Efficient primitives for deep learning,” *arXiv preprint arXiv:1410.0759*, 2014.

[26] A. Haidar, S. Tomov, P. Luszczek, and J. Dongarra, “Magma embedded: Towards a dense linear algebra library for energy efficient extreme computing,” in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2015, pp. 1–6.

[27] AMD, “ROCm - Open Source Platform for HPC and Ultrascale GPU Computing .” [Online]. Available: https://github.com/RadeonOpenCompute/ROCm

[28] J. Khan, P. Fultz, A. Tamazov, D. Lowell, C. Liu, M. Melesse, M. Nandhimandalam, K. Nasyrov, I. Perminov, T. Shah, V. Filippov, J. Zhang, J. Zhou, B. Natarajan, and M. Daga, “Miopen: An open source library for deep learning primitives,” 2019.

[29] S. Mittal and J. S. Vetter, “A survey of methods for analyzing and improving gpu energy efficiency,” *ACM Computing Surveys (CSUR)*, vol. 47, no. 2, pp. 1–23, 2014.

[30] Y. Jiao, H. Lin, P. Balaji, and W.-c. Feng, “Power and performance characterization of computational kernels on the gpu,” in *2010 IEEE/ACM Int’l Conference on Green Computing and Communications & Int’l Conference on Cyber, Physical and Social Computing*. IEEE, 2010, pp. 221–228.

[31] J. Guerreiro, A. Ilic, N. Roma, and P. Tomás, “Dvfs-aware application classification to improve gpgpus energy efficiency,” *Parallel Computing*, vol. 83, pp. 93–117, 2019.

[32] C. Constantinescu, I. Parulkar, R. Harper, and S. Michalak, “Silent data corruption—myth or reality?” in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 108–109.

[33] J. Guerreiro, A. Ilic, N. Roma, and P. Tomas, “Gpgpu power modeling for multi-domain voltage-frequency scaling,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 789–800.

[34] AMD, “HCC : An open source C++ compiler for heterogeneous devices.” [Online]. Available: https://github.com/RadeonOpenCompute/hcc

[35] ——, "HIP: C++ Heterogeneous-Compute Interface for Portability ." [Online]. Available: https://github.com/ROCm-Developer-Tools/HIP

[36] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 2001, pp. 14–25.

[37] D. M. Tullsen and J. S. Seng, "Storageless value prediction using prior register values," in *ACM SIGARCH Computer Architecture News*, vol. 27, no. 2. IEEE Computer Society, 1999, pp. 270–279.

[38] V. J. Reddi, M. S. Gupta, K. K. Rangan, S. Campanoni, G. Holloway, M. D. Smith, G.-Y. Wei, and D. Brooks, "Voltage noise: Why it's bad, and what to do about it," in *5th IEEE Workshop on Silicon Errors in Logic-System Effects (SELSE), Palo Alto, CA*. Citeseer, 2009.

[39] V. J. Reddi, S. Kanev, W. Kim, S. Campanoni, M. D. Smith, G.-Y. Wei, and D. Brooks, "Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2010, pp. 77–88.

[40] J.-P. Colinge *et al.*, *FinFETs and other multi-gate transistors*. Springer, 2008, vol. 73.

[41] J. H. Stathis, S. Mahapatra, and T. Grasser, "Controversial issues in negative bias temperature instability," *Microelectronics Reliability*, vol. 81, pp. 244–251, 2018.

[42] E. Cai, D. Stamoulis, and D. Marculescu, "Exploring aging deceleration in finfet-based multi-core systems," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–8.

[43] P.-F. Lu, K. A. Jenkins, T. Webel, O. Marquardt, and B. Schubert, "Long-term nbti degradation under real-use conditions in ibm microprocessors," *Microelectronics Reliability*, vol. 54, no. 11, pp. 2371–2377, 2014.

[44] K. J. Kuhn, M. D. Giles, D. Becher, P. Kolar, A. Kornfeld, R. Kotlyar, S. T. Ma, A. Maheshwari, and S. Mudanai, "Process technology variation," *IEEE Transactions on Electron Devices*, vol. 58, no. 8, pp. 2197–2208, 2011.

[45] S. Mittal, "A survey of architectural techniques for managing process variation," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–29, 2016.

[46] K. A. Bowman, S. G. Duvall, and J. D. Meindl, "Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration," *IEEE Journal of solid-state circuits*, vol. 37, no. 2, pp. 183–190, 2002.

[47] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[48] Tensorflow, "TensorFlow-Slim image classification model library." [Online]. Available: https://github.com/MachineLP/models/tree/master/research/slim

[49] G. E. Hinton, A. Krizhevsky, and I. Sutskever, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1106–1114, 2012.

[50] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[51] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[52] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," *arXiv preprint arXiv:1602.07261*, 2016.

[53] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[54] C. Holmes, D. Mawhirter, Y. He, F. Yan, and B. Wu, "Grnn: Low-latency and scalable rnn inference on gpus," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

[55] R. Kiros, Y. Zhu, R. R. Salakhutdinov, R. Zemel, R. Urtasun, A. Torralba, and S. Fidler, "Skip-thought vectors," in *Advances in neural information processing systems*, 2015, pp. 3294–3302.

[56] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 19–27.

[57] A. Radford, R. Jozefowicz, and I. Sutskever, "Learning to generate reviews and discovering sentiment," *arXiv preprint arXiv:1704.01444*, 2017.

[58] J. McAuley, R. Pandey, and J. Leskovec, "Inferring networks of substitutable and complementary products," in *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, 2015, pp. 785–794.

[59] W. B. Dolan and C. Brockett, "Automatically constructing a corpus of sentential paraphrases," in *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*, 2005.

[60] Max Woolf, “ReactionRNN.” [Online]. Available: https://github.com/minimaxir/reactionrnn

[61] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[62] Pandu Nayak. (2019, October) Understanding searches better than ever before. Accessed 18-October-2020. [Online]. Available: https://blog.google/products/search/search-language-understanding-bert/

[63] 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*, 2017, pp. 5998–6008.

[64] Google Research, “BERT.” [Online]. Available: https://github.com/google-research/bert