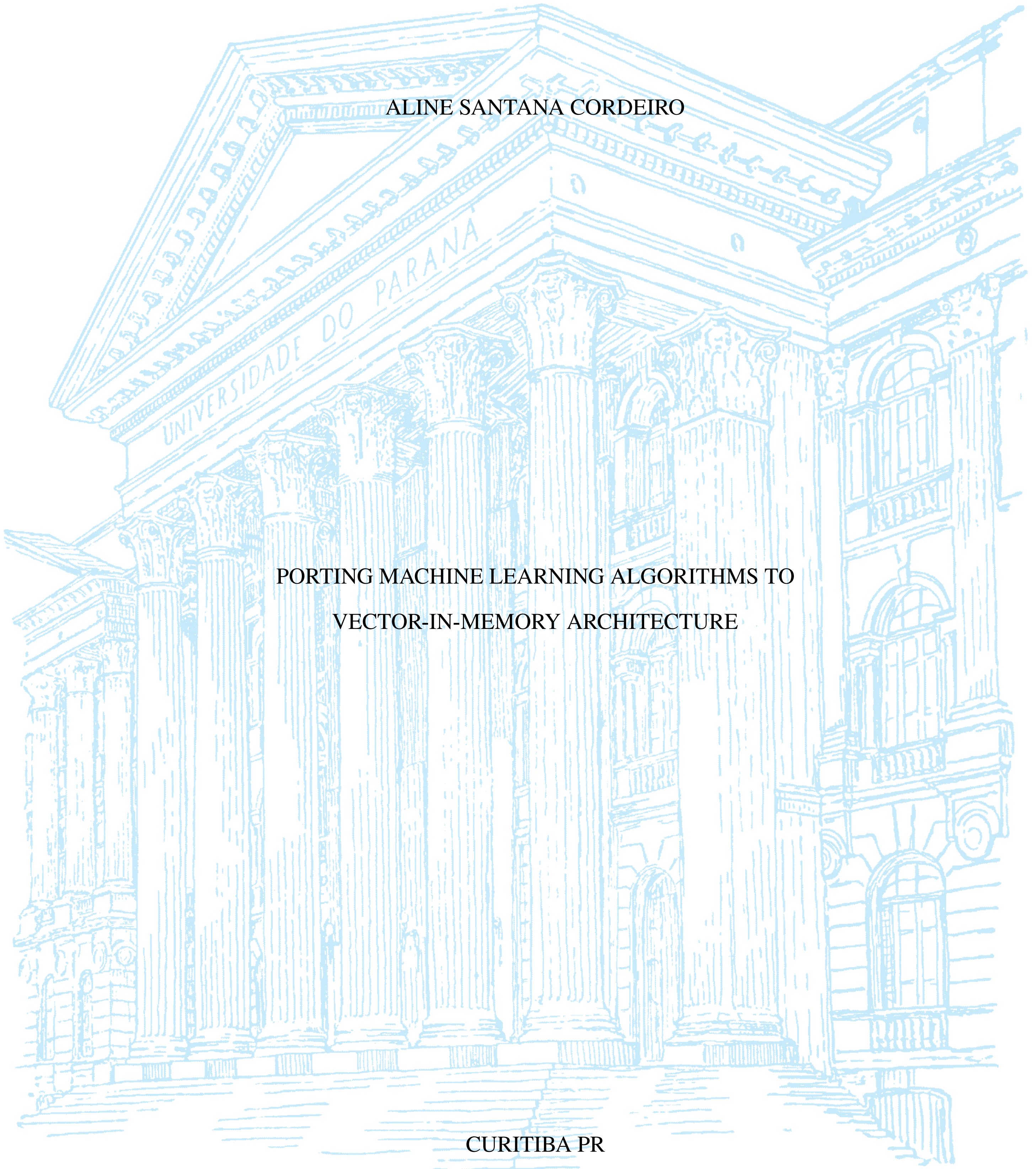
UNIVERSIDADE FEDERAL DO PARANÁ



ALINE SANTANA CORDEIRO

PORTING MACHINE LEARNING ALGORITHMS TO VECTOR-IN-MEMORY ARCHITECTURE

CURITIBA PR

2020

ALINE SANTANA CORDEIRO

PORTING MACHINE LEARNING ALGORITHMS TO VECTOR-IN-MEMORY ARCHITECTURE

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. Marco Antonio Zanata Alves.

CURITIBA PR

2020

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

C794p Cordeiro, Aline Santana
Porting machine learning algorithms to vector-in-memory architecture [recurso eletrônico] / Aline Santana Cordeiro. – Curitiba, 2020.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2020.

Orientador: Marco Antonio Zanata Alves.

1. Aprendizado do computador. 2. Algoritmos. 3. Sistemas de memória de computadores. I. Universidade Federal do Paraná. II. Alves, Marco Antonio Zanata. III. Título.

CDD: 006.32

Bibliotecária: Vanusa Maciel CRB- 9/1928

MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA -
40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da dissertação de Mestrado de **ALINE SANTANA CORDEIRO** intitulada: **Porting Machine Learning Algorithms to Vector-in-Memory Architecture**, sob orientação do Prof. Dr. MARCO ANTONIO ZANATA ALVES, que após terem inquirido a aluna e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 25 de Novembro de 2020.

Assinatura Eletrônica
25/11/2020 19:58:20.0
MARCO ANTONIO ZANATA ALVES
Presidente da Banca Examinadora (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
30/11/2020 07:50:32.0
LUIS CARLOS ERPEN DE BONA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
27/11/2020 10:12:16.0
EDUARDO ROCHA RODRIGUES
Avaliador Externo (IBM RESEARCH)

ACKNOWLEDGEMENTS

After almost 3 years of work and facing a harsh year of pandemic, I would like to thank Capes and Serrapilheira for funding and making possible my research with exclusive dedication during this period.

Likewise, I would like to thank my supervisor, Ph.D. Marco Antonio Zanata Alves, for all the support and guidance since my undergrad. I am very grateful that you have given me the opportunity and encouragement to start and finish this research.

Besides, I would like to acknowledge Ph.D. Paulo Cesar Santos from UFRGS, for his contributions to this dissertation proposal, the support, and discussions at the end of this work.

I also would like to thank all my colleagues from our laboratory, HiPES, who have supported me during this period. Especially MSc. Sairo Raoní dos Santos, who has been working with me on the same project and shared joys and sorrows.

In addition, I would like to thank my companion Rafael, my sister Deyse, my parents, and my friends for all the support and advice. You all have given me the opportunity in many ways of doing and finishing this work.

RESUMO

A Aprendizagem de Máquina surgiu por volta de 1960, com o foco na capacidade de aprendizagem do computador e, desde então, se tornou uma ferramenta útil para analisar a vasta quantidade de dados que é gerada em todos os campos da ciência nos dias de hoje. Ao longo dos anos, diversos algoritmos foram criados para analisar, reconhecer padrões e fazer previsões a partir de amostras de dados e, simultaneamente, a movimentação de dados dentro de sistemas computacionais ganhou foco devido ao seu alto impacto no tempo de execução e no consumo de energia. Nesse contexto, as arquiteturas de processamento próximo à memória surgiram como uma solução promissora para o processamento massivo de dados, reduzindo drasticamente a movimentação destes. Além das abordagens mais comuns para o problema, como *Central Processing Units (CPUs)* e *Graphic Processing Units (GPUs)*, também existem abordagens diferentes, como *Application Specific Integrated Circuits (ASICs)* e *Field-Programmable Gate Arrays (FPGAs)*. Esses aceleradores são opções interessantes para executar algoritmos de aprendizagem de máquina, no entanto, eles ainda apresentam problemas relacionados ao *Memory-Wall*, pois exigem movimentação de dados fora do chip entre a memória e os dispositivos de processamento e, como as soluções de processamento próximo à memória conectam a unidade de processamento ao dispositivo de armazenamento, elas reduzem os problemas originados pela movimentação de dados. Este trabalho avalia se é possível obter alto desempenho computacional para algoritmos de aprendizagem de máquina usando uma arquitetura de processamento próximo à memória que seja de propósito geral e que executa instruções vetoriais. Assim, é apresentada uma abordagem para executar alguns *kernels* de inferência como *k-Nearest Neighbors (kNN)*, *Multi Layer Perceptron (MLP)* e *Convolutional Neural Network (CNN)* usando a arquitetura de Vetor-em-Memória (VIMA), uma arquitetura de processamento próximo à memória que permite reutilização de dados e redução da latência de execução. A ideia é migrar esses algoritmos de aprendizagem de máquina com a Intrinsics-VIMA, uma biblioteca que emula o cojunto de instruções da VIMA e simula as aplicações usando um simulador orientado a traços para avaliar seu desempenho computacional e o consumo de energia. As contribuições deste trabalho são: (i) uma nova biblioteca Intrinsics que emula o conjunto de instruções da VIMA de forma fácil; (ii) ideias sobre como migrar algoritmos de aprendizagem de máquina usando Intrinsics-VIMA, e; (iii) a avaliação dos resultados dos algoritmos considerando o ambiente de simulação. Os resultados indicam acelerações de até 10× para o kNN, 11× para o MLP e 3× para a convolução ao executá-los na VIMA comparado com uma versão de alto desempenho do x86.

Palavras-chave: memórias inteligentes, processamento próximo à memória, aprendizagem de máquina, arquitetura vetorizada

ABSTRACT

Machine Learning (ML) emerged around 1960, focusing on the computer learning capacity. Since then, it became a handy tool to analyze the vast amount of data currently generated in every field of science. For this purpose, several algorithms were created to analyze, recognize patterns, and make predictions from data samples. Simultaneously, data movement inside computer systems gains more focus due to its high impact on time and energy consumption. In this context, the Near-Data Processing (NDP) architectures emerged as a prominent solution to massive data processing by drastically reducing data movement. Besides the most common approaches to the problem, such as Central Processing Units (CPUs) and Graphics Processing Units (GPUs), there are also different approaches, such as Application Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs). These accelerators are all exciting options to solve ML algorithms. Nevertheless, they still present problems related to the Memory Wall, as they still requiring off-chip data movement between the memory and the processing devices. As NDP solutions attach the processing unit to the storage device, they mitigate problems originated by data movement. This work evaluates whether it is possible to achieve high computational performance for ML algorithms using a general-purpose NDP architecture that operates on vector instructions. Thus, it presents an approach to execute inference kernels from k-Nearest Neighbors (kNN), Multi Layer Perceptron (MLP), and Convolutional Neural Network (CNN) algorithms using Vector-in-Memory Architecture (VIMA), an NDP architecture that allows data reuse and latency reduction. The idea is to port those ML algorithms with Intrinsics-VIMA. This library emulates VIMA Instruction Set Architecture (ISA), and simulate the applications using a trace-driven simulator to evaluate its computational performance and energy consumption. The contribution of this work are: (i) a new Intrinsics library that emulates VIMA ISA in an easy way; (ii) insights on how to migrate ML algorithms using Intrinsics-VIMA, and; (iii) results evaluation of the algorithms considering the simulation environment indicate speedups up to 10× for KNN, 11× for MLP, and 3× for convolution when executing near-data compared to a high-performance x86 baseline.

Keywords: smart-memories, near-data processing, machine learning, vector architecture

LIST OF FIGURES

2.1 Block diagram of an 3D-stacked memory 20
2.2 Block diagram of NDP architectures interconnection 20
2.3 HMC block diagram with 32 vaults with 16 banks each one 21
2.4 Architectural difference between HIVE and VIMA 22
2.5 3D-stacked memory module with VIMA architecture 23
2.6 Example of x86 assembly replacement 25

4.1 Sequence of steps to simulate an application 35
4.2 Von Neumann neighboring convolution 37
4.3 Moore neighboring convolution 37
4.4 Instance classification by votes 38
4.5 A neuron representation. 39
4.6 A representation of a neural network 39
4.7 Neural network output distribution at the beginning of the training 40
4.8 Moore neighboring convolution 42
4.9 The whole training dataset has to be available for each test instance. 42
4.10 Full utilization of a VIMA vector for training and test instances 42
4.11 VIMA vectors with training instances with 32 features and the respective labels . 43
4.12 Operation to apply a mask over a VIMA vector of 8 KB 43
4.13 Example of a multiplication between input layer and sets of weights 44
4.14 Example of a VIMA vector with four sets of weights 45

5.1 VIMA's speedup over x86 for kNN with 4096 instances. 51
5.2 VIMA's speedup over x86 for kNN with 8192 instances. 51
5.3 VIMA's speedup over x86 for kNN with 16384 instances 51
5.4 VIMA's speedup over x86 for kNN with 32768 instances 52
5.5 VIMA's speedup over x86 for kNN with 65536 instances 52
5.6 VIMA's speedup over x86 for MLP with 4096 instances. 53
5.7 VIMA's speedup over x86 for convolution 53
5.8 VIMA's energy consumption over x86 for kNN with 4096 instances 54
5.9 VIMA's energy consumption over x86 for kNN with 8192 instances 54
5.10 VIMA's energy consumption over x86 for kNN with 16384 instances. 55
5.11 VIMA's energy consumption over x86 for kNN with 32768 instances. 55
5.12 VIMA's energy consumption over x86 for kNN with 65536 instances. 55

5.13 VIMA's energy consumption over x86 for MLP with 4096 instances 56
5.14 VIMA's energy consumption over x86 for convolution 56

LIST OF TABLES

2.1 Bandwidths comparison 21

3.1 Strings used in search base 27
3.2 Inclusion/Exclusion criteria 28
3.3 Summary of correlated papers characteristics 29

4.1 Intrinsics-VIMA data types 36

5.1 Baseline and VIMA system configuration 47
5.2 kNN Memory Footprint approximation for VIMA 256B, 8KB and AVX512 . . . 49
5.3 MLP Memory Footprint approximation for VIMA 256B, 8KB and AVX512 . . . 49
5.4 Convolution Memory Footprint approx. for VIMA 256B, 8KB and AVX512 . . . 50

A.1 Table of Intrinsics-VIMA instructions 70
B.1 Table of selected papers 78

LIST OF ACRONYMS

AI	–	Artificial Intelligence
ALU	–	Arithmetic Logic Unit
API	–	Application Programming Interface
APU	–	Accelerated Processing Unit
ASIC	–	Application Specific Integrated Circuit
AVX	–	Advanced Vector Extensions
CNN	–	Convolutional Neural Network
CPU	–	Central Processing Unit
DDR	–	Double Data Rate
DIMM	–	Dual Inline Memory Module
DMA	–	Direct Memory Access
DNN	–	Deep Neural Network
DRAM	–	Dynamic Random Access Memory
FP	–	Floating-Point
FPGA	–	Field-Programmable Gate Array
FU	–	Functional Unit
GPU	–	Graphics Processing Unit
HBM	–	High Bandwidth Memory
HIVE	–	HMC Instruction Vector Extensions
HMC	–	Hybrid Memory Cube
HPC	–	High Processing Computing
ISA	–	Instruction Set Architecture
kNN	–	k-Nearest Neighbors

LLC	–	Last-Level Cache
LUT	–	Look-Up-Table
MAPLE	–	MAssively Parallel Learning/Classification Engine
MIMD	–	Multiple Instruction, Multiple Data
MIPS	–	Microprocessor without Interlocked Pipeline Stages
ML	–	Machine Learning
MLP	–	Multi Layer Perceptron
MMX	–	Multi-Media eXtensions
MRAM	–	Magnetoresistive Random-Access Memory
NDP	–	Near-Data Processing
NIM	–	Neuron In-Memory
NN	–	Neural Network
NoC	–	Network-on-Chip
OoO	–	Out-of-Order
OrCS	–	Ordinary Computer Simulator
PE	–	Processing Element
PIM	–	Processing-In-Memory
ReLU	–	Rectified Linear Units
RISC	–	Reduced Instruction Set Computer
RRAM	–	Resistive Random-Access Memory
RVU	–	Reconfigurable Vector Unit
SIMD	–	Single Instruction Multiple Data
SiNUCA	–	Simulator of Non-Uniform Cache Architectures
SRAM	–	Static Random Access Memory
SSE	–	Streaming SIMD Extensions
TLB	–	Translation Look-aside Buffer

TSV	–	Through-Silicon Via
VIMA	–	Vector-in-Memory Architecture

CONTENTS

1 INTRODUCTION . . . 14
2 BACKGROUND . . . 17
2.1 MACHINE LEARNING . . . 17
2.2 COMPUTING PERFORMANCE . . . 18
2.3 NEAR-DATA PROCESSING. . . . 19
2.4 VECTOR-IN-MEMORY ARCHITECTURE. . . . 21
2.5 INTRINSICS LIBRARIES . . . 24
2.6 ORDINARY COMPUTING SIMULATOR . . . 24
3 RELATED WORK USING SYSTEMATIC MAPPING . . . 26
3.1 SYSTEMATIC MAPPING METHODOLOGY . . . 26
3.2 STATE-OF-THE-ART . . . 30
3.2.1 NDP Approaches with Full Cores . . . 31
3.2.2 NDP Approaches with General-Purpose Cores. . . . 32
3.2.3 NDP Approaches with Embedding Specific-Purpose Cores . . . 32
3.2.4 DRAM and PIM Approaches . . . 33
3.2.5 Conclusions on Related Work. . . . 34
4 MACHINE LEARNING CODE PORTABILITY . . . 35
4.1 INTRINSICS-VIMA . . . 35
4.2 OVERVIEW: MACHINE LEARNING KERNELS . . . 37
4.2.1 Convolution Basics . . . 37
4.2.2 k-Nearest Neighbors (KNN) Basics. . . . 38
4.2.3 Multi-Layer Perceptron (MLP) Basics . . . 38
4.3 CODE PORTABILITY: VIMA . . . 41
4.3.1 Convolution Migration . . . 41
4.3.2 k-Nearest Neighbors (KNN) Migration . . . 42
4.3.3 Multilayer Perceptron (MLP) Migration . . . 44
5 EXPERIMENTAL EVALUATION OF VIMA . . . 47
5.1 METHODOLOGY AND SIMULATION SETUP . . . 47
5.2 BEST CONDITIONS TO ACHIEVE HIGH PERFORMANCE. . . . 48
5.2.1 k-Nearest Neighbors. . . . 48
5.2.2 Multilayer Perceptron . . . 49
5.2.3 Convolution . . . 49
5.3 EXECUTION TIME RESULTS . . . 50
5.4 ENERGY RESULTS . . . 54

6 FINAL CONSIDERATIONS 57
7 CONCLUSION 59
REFERENCES 60
APPENDIX A – TABLE OF INTRINSICS-VIMA INSTRUCTIONS 69
APPENDIX B – TABLE OF MAPPING STUDY 77
APPENDIX C – KNN ALGORITHM WITH AVX512 81
APPENDIX D – MLP HIDDEN LAYER ALGORITHM WITH AVX512 . 82
APPENDIX E – MLP OUTPUT LAYER ALGORITHM WITH AVX512 . 83
APPENDIX F – CONVOLUTION ALGORITHM WITH AVX512 84
APPENDIX G – KNN ALGORITHM WITH 8KB VIMA VECTOR 85
APPENDIX H – KNN ALGORITHM WITH VIMA 8KB VECTOR 86
APPENDIX I – MLP HIDDEN LAYER ALGORITHM WITH VIMA 8KB VECTOR 87
APPENDIX J – MLP HIDDEN LAYER ALGORITHM WITH VIMA 8KB VECTOR 88
APPENDIX K – CONVOLUTION ALGORITHM WITH VIMA 8KB VECTOR 89

1 INTRODUCTION

Over the last two decades, digital data has increased significantly due to the increase of digital systems usage, which executes different kinds of transactions such as registration, creation, sharing, and downloading of information. For example, in 2011, the amount of created and replicated data surpassed 1.8 zettabytes, and predictions expect that this amount will grow 9× every five years. In this scenario, we can expect digital data volume to double every two years, reaching 40 trillion gigabytes at this year of 2020 (Gantz and Reinsel, 2011, 2012).

Those massive digital transactions carry lots of information about people's behavior in different aspects. All these data can be interesting for scientists and corporations to analyze to understand people's needs and develop personal products and services. Humans cannot analyze and understand this massive amount of data, firstly because of the volume and time required, secondly, due to its complexity. Thus, due to the increased processing capacity available in current computers, Machine Learning (ML) has gained popularity and became a useful tool to automate the analysis of massive amounts of data (Krizhevsky et al., 2012; Rakotomamonjy, 2003; Gardner and Dorling, 1998; Peterson, 2009; Dietterich, 2000).

ML was formalized in 1959 (Samuel, 1959) and focuses on systems that can learn and adapt to environmental changes without being explicitly programmed to it. Such algorithms rely on observations and sometimes learning from external examples to become capable of making their own decisions, identifying patterns, and making future decisions (Alpaydin, 2009).

Despite existing complex ML applications that are efficient in common architectures (Chen and Guestrin, 2016), some are memory and computationally intensive. Thus, experiments made in the last decades were mostly with small data sets, which changed with the current computer systems. Nevertheless, general-purpose computers and their ever-increasing performance still present severe bottlenecks in terms of the execution time of ML algorithms when dealing with real-world size problems (Boroumand et al., 2018). In this way, the most common implementations rely on accelerators and specific-hardware, such as Application Specific Integrated Circuit (ASIC), full Central Processing Units (CPUs), Field-Programmable Gate Arrays (FPGAs) (Nurvitadhi et al., 2016; Kara et al., 2017) and Graphics Processing Units (GPUs) (Gao et al., 2017; Ahn et al., 2016; Nair et al., 2015), which provide reasonable solutions due to their high computational capacity, allowing data parallelism and higher performance. Prominent designs based on simple vector units (e.g.,Functional Units (FUs)) (Alves et al., 2016; Santos et al., 2017; Oliveira et al., 2017a; Santos et al., 2018), also enable the highest energy efficiency while meeting the required constraints regarding the area and power (Lima et al., 2018).

Although all these alternatives can explore massive parallelism to execute ML algorithms, except for solutions implemented near memory, data movement between memory and the processor unit is still a bottleneck for data-intensive computations. Such a bottleneck is well known as memory-wall (Wulf and McKee, 1995). These previously mentioned devices rely on off-chip data transfer through interconnections (Ren, 2011; Sukhwani et al., 2012; Thoma et al., 2013). The memory-wall limitation is inherent to contemporary computer system designs, where the memory hierarchy can mitigate some of the performance drawbacks. However, in terms of energy and latency, it is not sufficient (Hashemi et al., 2016; Qureshi et al., 2007b,a).

Observing that data movement consumes as high as 60% of the total system energy (Boroumand et al., 2018), Near-Data Processing (NDP) and Processing-In-Memory (PIM) have emerged as promising solutions for the memory-wall problem, with the idea of integrating processor and memory in the same chip (Nowatzyk et al., 1996; Patterson et al., 1997b,a; Elliott

et al., 1999b). These ideas emerged as a product in the last few years, such as Hybrid Memory Cube (HMC) and High Bandwidth Memory (HBM) (Hybrid Memory Cube Consortium, 2014; Kim and Kim, 2014). Such products take advantage of 3D integration technology to integrate processing logic and memory in the same chip, mitigating data movement and, consequently, reducing data latency and increasing processing and memory performance, achieving high data parallelism (Pugsley et al., 2014; Oliveira et al., 2017b).

Therefore, this dissertation proposes an alternative to achieve high-performance computing for ML applications, but still using a general-purpose architecture. The main idea is to present the benefits of migrating the kernel of three well-known ML algorithms (k-Nearest Neighbors (kNN), Multi Layer Perceptron (MLP), and Convolutional Neural Network (CNN)) to an NDP design capable of large-vector operations named Vector-in-Memory Architecture (VIMA). VIMA is inspired by HMC Instruction Vector Extensions (HIVE) (Alves et al., 2016) to provide a complete environment for NDP. It is a module attached to a 3D-stacked memory, composed of an instruction sequencer, vector FUs, and the most crucial component, a small cache memory. In this way, VIMA can reduce data movement between host processors and main memory, increasing overall efficiency and performance by executing vector operations and reusing data near memory.

These algorithms were chosen due to their usability to solve different types of ML problems and because of the code structure used to implement them, which is vastly used in computing and allows data reuse during execution. Besides, these algorithms were evaluated with different sizes of datasets to observe their behavioral changes in VIMA. These characteristics are relevant here since we expect a higher performance from VIMA considering bigger datasets.

To allow this migration, we developed Intrinsics-VIMA, a vector-designed C/C++ library extension (Cordeiro et al., 2017). Intrinsics-VIMA facilitates the writing of codes for VIMA and similar NDP architectures, enabling the simulation and evaluation of new algorithms with reduced programming effort. As it is an x86-based library, the developer can use it like any other C/C++ library, enabling compilation, execution, and debugging.

The Intrinsics-VIMA library allows the generation of simulation traces to be consumed and evaluated by Ordinary Computer Simulator (OrCS), an in-house trace-driven simulator adopted to generate architectural results related to each application. The simulation traces are generated by an instrumentation tool, which analyzes the x86 binary code of the applications and introduces assembly VIMA instructions to the trace, when necessary. The simulator correctly interprets these assembly VIMA instructions.

Overall, in this dissertation, we present the evaluation of VIMA, a new NDP architecture, which mainly allows vector execution and data reuse, with vastly used algorithms of ML field. However, we do not intend to propose or evaluate ML algorithms, but on the architecture performance. Moreover, we also provide insights on how to migrate these algorithms to VIMA, considering the usage of large vector units and adjusting the algorithm to make better use of these units.

Our experimental results comparing the x86-only approach to the NDP migration show substantial improvements on execution time up to 10× for kNN, 11× for MLP, and 3× for convolution. Additionally, VIMA reduces energy consumption by up to 7× for kNN, 8× for MLP, and 3× for convolution.

In this dissertation, we present the following main contributions:

- We provide insights on how to migrate ML algorithms to a NDP architecture based on large vector units, showing benefits from NDP in this context.

- We extend and use an NDP intrinsics library that supports validation of NDP architectures based on large vectors.

The remaining of this document is organized as follows: Chapter 2, explains basic concepts for a better understanding of the proposal, discussing ML, NDP, OrCS, and Intrinsics libraries; Chapter 3 explains the research methodology and presents related work; Chapter 4 presents the main proposal of this work and expectations about it; Chapter 5 exhibits the obtained results related to speedup and energy efficiency and, finally; Chapter 7 brings final thoughts and future work.

2 BACKGROUND

Among the related issues, in this chapter we detail ML, computational limitations, NDP, VIMA, OrCS, Intrinsics libraries, and a trace generator using Pin.

2.1 MACHINE LEARNING

ML emerged in 1959 as a subfield of Artificial Intelligence (AI) and is concerned with studying the learning capacity of the firsts digital computers, developed around 1945. Those machines were capable of solving simple operations, such as addition, subtraction, and multiplication. However, researchers wanted to find a way to make those machines also "intelligent", a system that could learn by itself, without any programmer intervention. In this way, they strove to combine different abilities in digital computers, such as psychology, mathematics, philosophy, linguistics, and statistics, to enable it to analyze data and make its conclusions.

The main idea of ML is to enable a computer with no specific algorithm for a specific task to make correct decisions about it. In this way, the programmer must only input data and its correspondent labels when requested to a computer to its learning. The computer shall trace patterns used to classify newer samples or make a regression by predicting specific values. These kinds of tasks represent supervised learning techniques. Other techniques classified as unsupervised learning (Géron, 2019) relies on techniques such as: data clustering, dimensionality reduction, anomaly detection, or association rule learning. Another appealing paradigm is reinforcement learning, which allows progressive learning and can be applied in different fields of study, such as game theory, multi-agent systems, and statistics.

The specialist must add labels on the training dataset for supervised learning. The idea is to have an input instance and the desired output value to train the algorithm accurately. One of the most known techniques is classification, where an algorithm must be trained with a dataset to create a mathematical model representing that learning. Thus, during the training, the algorithm must frequently update a set of learning parameters considering each input's relevant features and its given label. After finishing the training phase, there will hardly be ideal values for these parameters to make the model accurate, so we must consider that there will be optimal values that will allow this. Besides, we consider a model accurate if it classifies the vast majority of the instances in a training dataset correctly. For some algorithms, such as Neural Networks (NNs) and Deep Neural Networks (DNNs), a bias is used along with the learning parameters to adjust the output better, and usually, the training phase is executed more than once. Each iteration in this process in the training dataset is called an epoch. Overall, the main objective of the training step is to reduce classification errors in the training dataset.

There is no label in the training dataset for unsupervised learning since these algorithms can group data in clusters considering specific attributes with undetected patterns that can make them similar between themselves. Thus, dimensionality reduction can simplify data without losing relevant information by extracting each data point's most relevant features. Anomaly detection algorithms can identify frauds and anomalies in datasets by detecting statistical outliers in the dataset. Association rule learning can analyze data to associate patterns that occur together and can be indirectly related. Finally, the trained model can be predictive or descriptive (Alpaydin, 2009; Géron, 2019).

Unlike supervised learning, reinforcement learning is a technique of progressive learning that does not require input labels or even rely on mathematical models with approximation

functions. Instead, it focuses on finding optimal solutions considering a specific algorithm and an environment. In other words, an agent in an environment has a goal to reach and, to do, so it has to take action. Each action has a reward linked to it and leads the agent to a different state in this environment. Thus the objective is to maximize the cumulative sum of the rewards received during the taken actions to reach this goal. These actions are repeated successively since it has to be taken for each state until the agent reaches its goal (Mitchell, 1997).

After the training phase, the validation occurs. If the model is not accurate enough, the developer must train the model again, considering different characteristics, parameters, and a more representative training dataset (Russell and Norvig, 2016; Alpaydin, 2009). Depending on these algorithms' final purpose, the programmer shall provide other information or rules available to achieve accurate results. However, the programmer should not interfere with the algorithm's learning process when calculating the model parameters, as it is expected that the ML algorithm learn by itself.

Once the model is validated, the inference phase can start. Here, the ML algorithm will make decisions for each sample from a set of test samples, considering the mathematical parameters defined during the training phase. Both training and inference are computation-intensive tasks. The training relies on latency since it depends on massive operations over a massive set of training instances during multiple epochs to define the model parameters. On the other hand, inference relies on high throughput to classify a stream of instances, representing real-time applications, making this phase more critical than training.

Thus, the training phase depends on robust architectures to achieve high-performance computing. While the inference can be executed even in embedded systems with limited hardware resources, such as FPGA and autonomous vehicles (McDanel et al., 2017; Qiu et al., 2016; Tian et al., 2018), which means that the model must be trained once to be used multiple times by multiple devices.

Nowadays, it is easier to employ ML in different tasks due to different frameworks available in high-level languages (Raschka, 2015) and a vast amount of accessible datasets used to train and test the models. Thus, in the last few years, we could see a flourishing of applications for different fields such as object identification tools, voice recognition, text context analysis. Researchers are using such algorithms in diverse areas as genetics (Libbrecht and Noble, 2015), cancer prognosis (Kourou et al., 2015), autonomous vehicles (Kuderer et al., 2015), and facial expressions (Bartlett et al., 2004). These are examples of tasks that have the potential to bring a great advance in science for society.

For the remain of this dissertation, we consider to work with three ML kernels: kNN, MLP, and CNN. We implemented naive versions of each algorithm, and we intend to show just the most significant computation part of each algorithm since the idea is to evaluate them in VIMA. For example, the kernel of CNN is the convolution part only, while the ML is a plain NN where we implemented the inference part only. Together, they are a simplified version of CNN, which results give us an estimate of the whole algorithm. Further explanations of these algorithms will be present in Chapter 4.

2.2 COMPUTING PERFORMANCE

Although ML offers a series of exciting solutions, it is not always possible to make proper use of this tool in a general-purpose processor. When working with a massive amount of data, complex ML algorithms can be computationally consuming, resulting in low performance. This low performance is due to the Von Neumann bottleneck, in which CPU and Dynamic Random Access

Memory (DRAM) communication restricts the computation capacity, i.e., the speed to acquire data from DRAM is significantly slower than the data processing speed (Shen and Lipasti, 2013).

Over the years, architects proposed different approaches to mitigate this bottleneck. The firsts attempts to improve performance emerged around 1980 with the cache memory, which inserts one level of a smaller and faster memory between CPU and memory. In this way, the cache will store recently used data and fetch it to the CPU faster than the DRAM (Smith, 1982; Shen and Lipasti, 2013).

Around 1990, superscalar in-order and Out-of-Order (OoO) processors emerged with a deeper pipeline, enabling fetching and executing more than one instruction per cycle. In the execution step, instructions with no dependencies can be executed before its predecessors, improving performance aggressively. By the same time, Single Instruction Multiple Data (SIMD) instructions became famous in the '90s with the Multi-Media eXtensions (MMX) Intel instruction set, followed by Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) (20 years later), enabling the execution of vector instructions. In other words, they were implementing a specific operation over a set of operands in a few cycles (Shen and Lipasti, 2013).

According to Moore's Law, every 18 months (and currently every 24 months), the number of transistors shall double, resulting in increased processing capacity (Moore, 1998; Moore et al., 1975). Initially, architects made these technological integration improvements to convert in performance gains directly. However, this is no longer true nowadays due to problems such as memory-wall (Wulf and McKee, 1995) and dark silicon (Esmaeilzadeh et al., 2011). In summary, the CPU has dramatically improved over the years, but memory improvements are happening at a much slower pace (Efnusheva et al., 2017).

Besides, despite the improvements in general computing performance, energy consumption is still a problem, as 60% of the energy spent during execution refers to the data movement between processor and memory (Boroumand et al., 2018). Thus, the energy problem will remain even when executing ML algorithms in accelerators and specific-purpose hardware to achieve higher performance, as these devices, such as FPGA and GPU, will still communicate with CPU and memory through the bus (Sukhwani et al., 2012; Thoma et al., 2013).

2.3 NEAR-DATA PROCESSING

NDP dates back to the 1990s (Patterson et al., 1997b; Elliott et al., 1999a), when the industry was unable to integrate DRAM and logic cells on the same die. However, with the advent of 3D integration, NDP has reemerged as a viable solution. Architects came up with this new architecture concept to mitigate data movement between memory and processor. Thereby, processing occurs in the same chip as the memory, as illustrated in Figure 2.1. This type of architecture improves performance and energy consumption while presenting high parallelism, thus ensuring low average latency during high pressure in memory. Such architectures are ideal for streaming and parallel applications, graphics, High Processing Computing (HPC), and networking. Generally, any application with coalescent memory accesses can benefit from it.

The most well-known 3D-stacked memory commercial examples are HMC (Hybrid Memory Cube Consortium, 2014) and HBM (Jun et al., 2017). Both architectures focus on the NDP concept, but some details differ between these. On the one hand, Micron released HMC in 2011 (Pawlowski, 2011b) with a completely new hardware and protocol specification. HMC used a 3D architecture to embed memory controller and processing logic near-data. It uses high-frequency serial links formed by full-duplex lanes, enabling data transmissions with low interference, to connect HMC to the processor (Thanh-Hoang et al., 2014). HMC is illustrated in Figure 2.2(a). Nevertheless, HMC requires a new protocol for memory control, which also

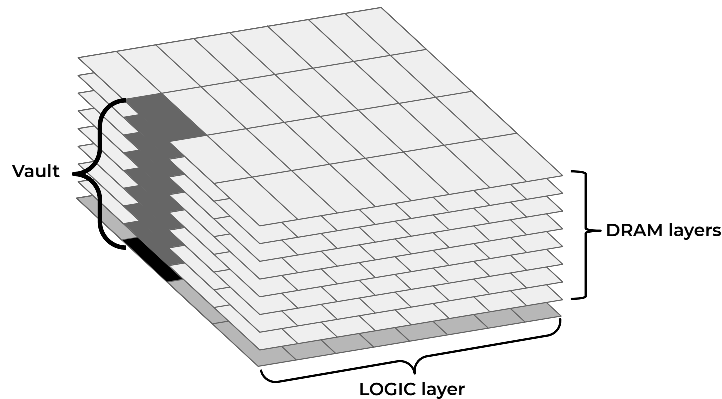
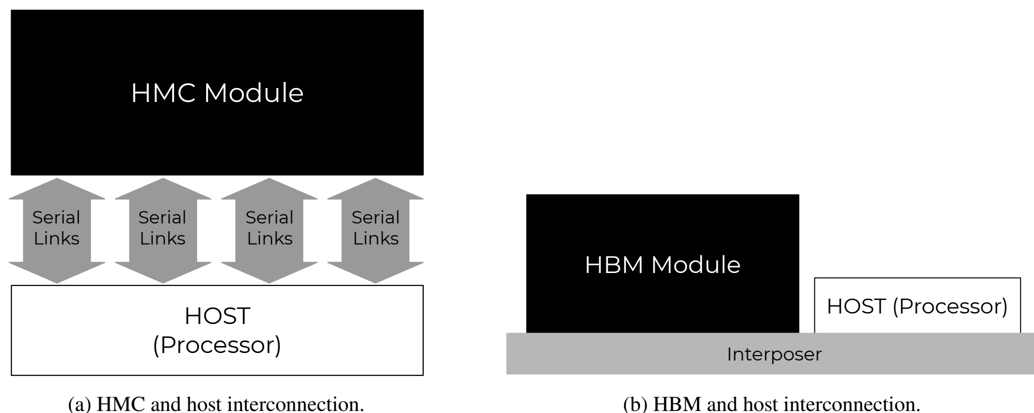


Figure 2.1: Block diagram of an 3D-stacked memory.

required changes on the processor side. On the other hand, HBM is JEDEC compliant since 2013 and has a well-known specification. Different from HMC, HBM is called a 2.5D architecture due to its integration system, which uses a silicon layer, called interposer, to attach both HBM memory and host CPU or GPU in the same die, as illustrated in Figure 2.2(b).



(a) HMC and host interconnection.

(b) HBM and host interconnection.

Figure 2.2: Block diagram of NDP architectures interconnection.

Overall, both architectures share several design characteristics. Both are compound by up to 8 stacked layers of DRAMs having a base that is a logic layer. Such a logic layer can integrate a processor to operate elements inside the memory. The 3D-stacked memory module is logically partitioned in up to 32 vaults, similar to the memory channel concept. Each vault is composed of up to 16 independent DRAM banks, distributed among DRAM layers, connected through Through-Silicon Vias (TSVs) (Olmen et al., 2008), as illustrated in Figure 2.3. NDP architectures can hide their data access internal latency due to their high bandwidth provided by internal parallelism (Hybrid Memory Cube Consortium, 2013; Jeddeloh and Keeth, 2012; Pawlowski, 2011a) achieved by 3D integration technology together with the 32 vaults.

Compared to Double Data Rate (DDR) memory technology, NDP devices require, on average, the same voltage level. However, 3D memories can achieve higher memory bandwidth, reaching up to 410 GB/s in the latest version (JESD235C) (Transcend, 2014; AMD, 2015), as defined in Table 2.1. It is also possible to observe that 3D-stacked memories have higher energy efficiency than DDR (Hrusca, 2015) due to its smaller row buffers and modified open row policy.

Finally, NDP can mitigate the memory-wall problem in contrast to CPU, GPU, and FPGA, which all require time and energy inefficient off-die (or off-chip) data transfers, by eliminating data movements from the memory hierarchy.

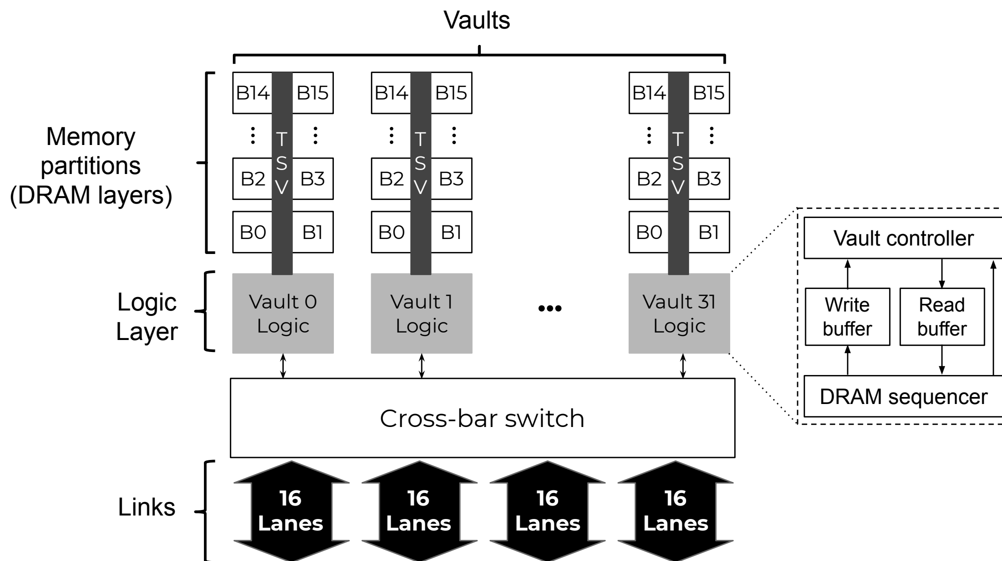


Figure 2.3: HMC block diagram with 32 vaults with 16 banks each one. Adopted from (Hybrid Memory Cube Consortium, 2013).

Table 2.1: Bandwidths comparison.

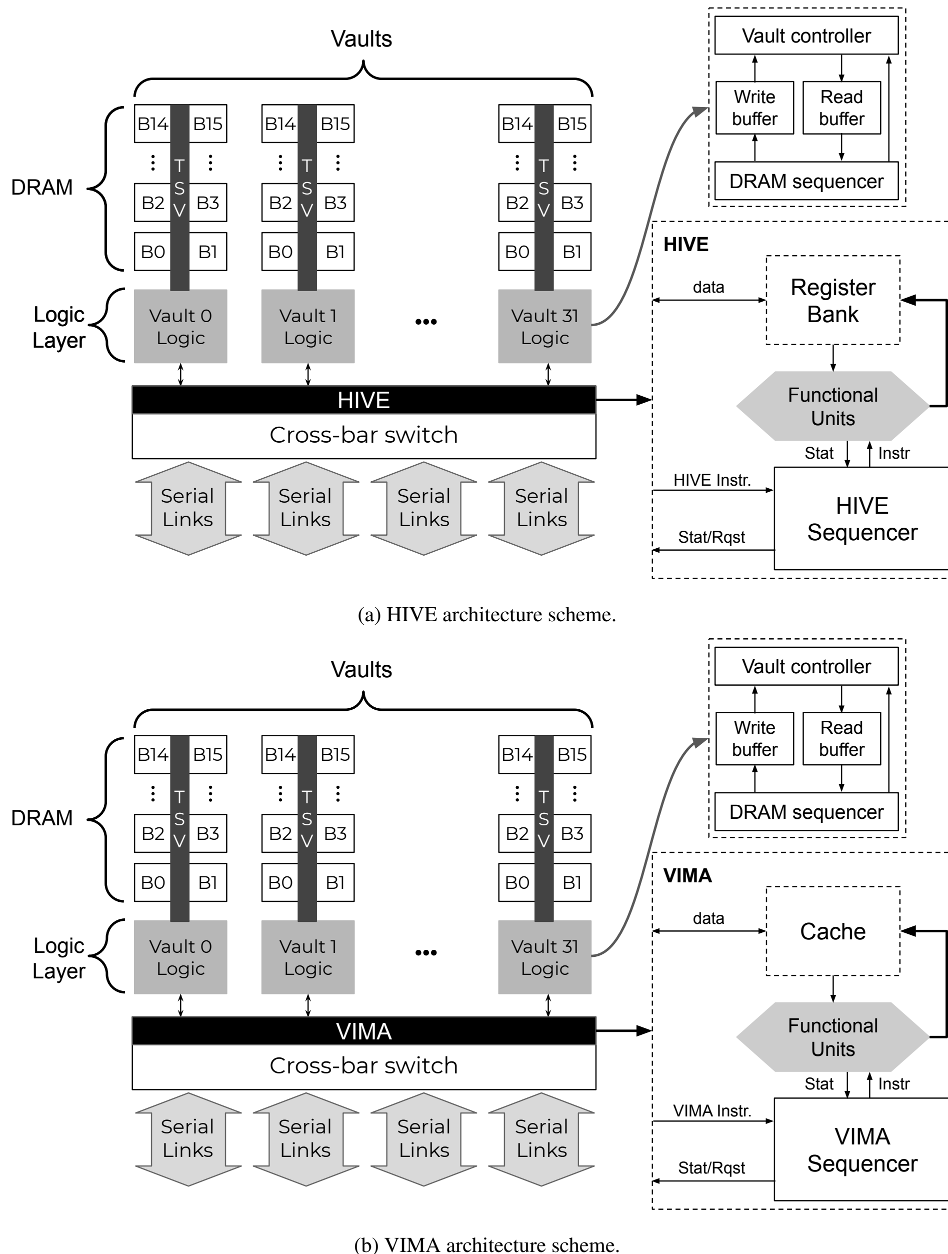
Memory	Bandwidth	Voltage	Speed (data rate/pin)	Energy efficiency	Jedec standard
DDR	3.2 GB/s	2.6 V	0.4 GT/s	257.13 pJ/b	yes
DDR2	6.4 GB/s	1.8 V	0.8 GT/s	121.44 pJ/b	yes
DDR3	14.9 GB/s	1.5 V	1.8 GT/s	64.70 pJ/b	yes
DDR4	25.6 GB/s	1.2 V	3.2 GT/s	38.67 pJ/b	yes
DDR5	41.6 GB/s	1.1 V	5.2 GT/s	N.A.	yes
HMC	320 GB/s	1.2 V	2.5 GT/s	10.82 pJ/b	no
HBM1	128 GB/s	1.3 V	1.2 GT/s	N.A.	yes
HBM2	256 GB/s	1.3 V	2.0 GT/s	N.A.	yes
HBM2 2018	310 GB/s	1.2 V	2.4 GT/s	N.A.	yes
HBM2e	410 GB/s	1.2 V	3.2 GT/s	N.A.	yes

2.4 VECTOR-IN-MEMORY ARCHITECTURE

The main idea of this work is to evaluate ML algorithms in an NDP general-purpose processing that enables vector operation and does not require a full processor integration near data. In this context, HIVE enables the execution of large vector instructions that obtain data in parallel from the independent memory vaults inside a 3D-memory. It also includes vector extensions to the processor's Instruction Set Architecture (ISA) to control the near-data vector units. This design does not require any complex instruction fetch or decode unit implementation inside the memory. HIVE (Alves et al., 2016) was adopted as a model to define VIMA, that we will use in this dissertation.

VIMA is similar to other NDP approaches, which obtain data from several independent memory vaults in parallel (Alves et al., 2016; Santos et al., 2017; Tomé et al., 2018). The main difference is that VIMA replaces HIVE's register bank with a data cache memory of a similar size (i.e., 64 KB), which maintains high performance while providing transparency and high flexibility for programmers, as depicted in Figure 2.4. VIMA explores the data access parallelism inherent

to NDP architecture, and its cache enables fast reuse of vectorized data within the memory. Both HIVE and VIMA proposals support ARM NEON Integer and Floating-Point (FP) instructions, operating over vectors of 8 KB of data, which fetch data over the 32 channels (vaults) in parallel.



(a) HIVE architecture scheme.

(b) VIMA architecture scheme.

Figure 2.4: Architectural difference between HIVE and VIMA.

The main physical addition of VIMA compared to related work Alves et al. (2016); Santos et al. (2017) is a small cache memory that enables data-reuse of data vectors. One could perceive this as a minor change, but VIMA enables significant improvements due to its new operation rationale, such as improved data re-usage, easy-to-program interface, precise exceptions, extensible design, multi-threading,all discussed in the next sub-sections. At the same time, it maintains most of the performance improvements compared to any NDP strategy.

Similar to SSE, AVX, and NEON instruction sets, VIMA also needs an ISA extension, which must be used in the code by the programmer and is explained further, in section 2.5. VIMA instructions must be inserted in the application during compilation and pass through the processor

as a conventional memory instruction. However, they bypass the cache memory hierarchy, being sent directly to the 3D-stacked memory chip.

VIMA instructions operate over data vectors of 256 B and 8 KB. For instance, for a VIMA cache memory of 64 KB, it may store 32 vectors of 256 B, or 8 vectors of 8 KB. Beyond the cache memory, a set of vector FUs, and an instruction sequencer are added to VIMA, as depicted in Figure 2.5. Thus, as soon an instruction is sent to VIMA, it must wait in the sequencer until its required data is fetched from the 3D-stacked memory and become available in VIMA's cache memory. Then, the instruction executes in the vector FUs, and, as soon as it finishes, a signal is sent to the CPU, informing its execution status. If the execution is successful, the processor can commit the instruction. Otherwise, the pipeline must be flushed, and an exception is raised. VIMA executes instructions in-order to accomplish precise exceptions.

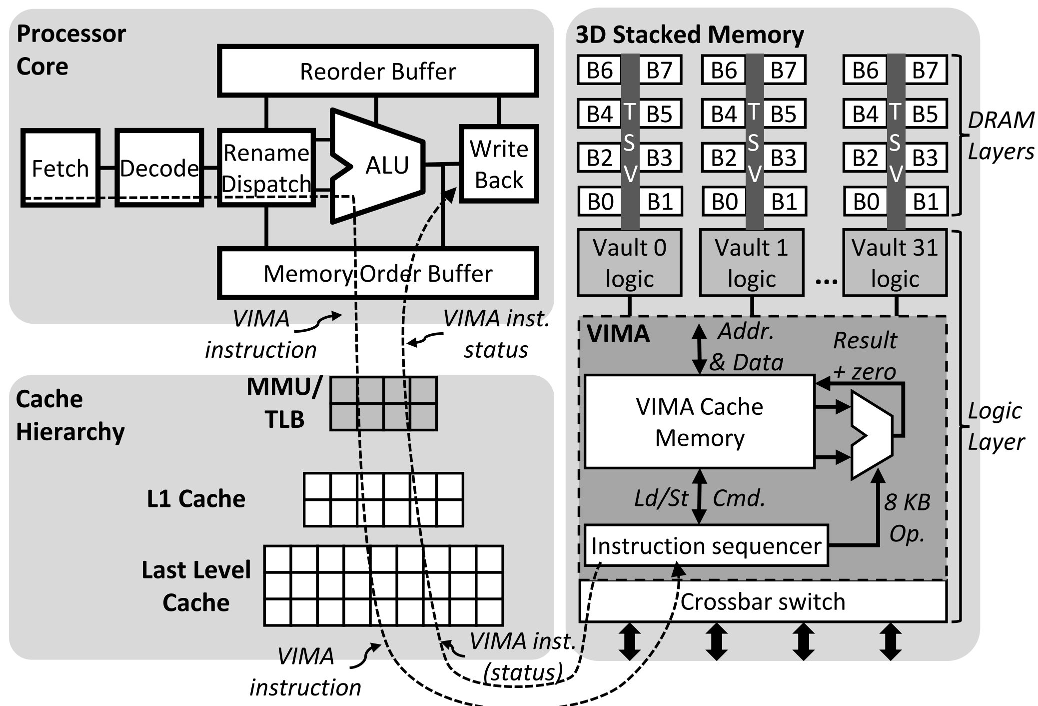


Figure 2.5: 3D-stacked memory module with VIMA architecture.

As the CPU dispatches the next VIMA instruction after committing the previous one, it has to deal with precise exceptions. Thus, it is possible to observe some negative impacts on VIMA. Firstly, an 8 KB vector enables the full parallelism of a 3D-stacked memory chip with 32 vaults and at least 8 banks per vault. However, 256 B vectors perform 74% worse than 8 KB in terms of execution time, on average. This performance decay happens because smaller vector sizes in VIMA, are unable to fully use the memory's internal parallelism. Secondly, precise-exceptions impose an in-order execution for VIMA instructions. Accordingly to our evaluations, these pipeline bubbles degrade between 2% and 4% the execution time.

VIMA maintains most of the high-performance of any NDP strategy while enabling significant improvements, such as improved data re-usage, easy-to-program interface, precise-exceptions, multi-threading, and extensible design. VIMA is flexible enough to allow changes in data vectors size, which may reduce or increase the parallelism inside the memory.

2.5 INTRINSICS LIBRARIES

We developed Intrinsics-VIMA library to enable quick and easy development of programs using the NDP instructions. It is inspired by Intel Intrinsics (Lomont, 2011), a library available in C language with a set of routines. When a program calls a routine from this library, it embeds its internal assembly x86 code directly in the compiler to optimize the execution. The assembly code generated during the compilation of a C code with the same functionalities, but without calling Intel intrinsic functions, might not be the same (Coorporation, 2009). Thus, these routines allow low-level code optimization, including code vectorization, known as SIMD instructions. Nevertheless, the vendors create and distribute intrinsic libraries accordingly to the ISA extension available inside each processor. Intel intrinsics are typically used by expert programmers that want to obtain most of the processor's ISA performance.

VIMA is an ISA extension that is not present in real-world processors yet. Thus, our Intrinsics-VIMA is a library composed of routines that reproduce the behavior of VIMA, using x86 instructions. Intrinsics-VIMA was developed in C/C++, allowing programmers to write, compile, execute, and debug code even for a non-existing architecture, since it is possible to simulate it in the x86 environment, ensuring the code correctness. Whenever the code is reliable, we can use it to generate simulation traces. At this step, the trace generator converts the routines into VIMA instructions that our simulation environment can interpret and simulate. We provide more details regarding the trace generator in Section 2.6.

We already developed other Intrinsics libraries for different target architectures, such as Intrinsics-HMC (Cordeiro et al., 2017) and Intrinsics-MIPS. These Intrinsics libraries are easy to develop, test, and extend to different architectures. Besides, all the Intrinsics libraries are available in GitHub [1]. Intrinsics-VIMA was developed for this work and will be fully detailed in Chapter 4.

2.6 ORDINARY COMPUTING SIMULATOR

During the evaluation of new processor architectures, simulation represents a viable and affordable solution for designers. It happens because the system to be evaluated is too complex to be handled by analytic models, and highly expensive to be prototyped (Jain, 1990). Thus, most computer architects use simulation tools. In contrast to full-system simulation, trace-driven simulators do not require real execution of the application instructions. Such simulators only consider the behavioral details (algorithmic) and microarchitectural latencies for the given traced instructions. These simulators use execution traces of real applications formed by one or multiple files containing the flow of instructions observed during the program execution. These traces can be generated manually by researchers or automatically by binary instrumentation tools.

In this work, we used OrCS, a cycle-accurate, trace-driven simulator, based on the x86 architecture, able to execute the ISAs x86_32, x86_64, and now, VIMA. OrCS is a simplified version of Simulator of Non-Uniform Cache Architectures (SiNUCA) (Alves et al., 2015; Alves, 2014). OrCS has a Trace Generator that automatically generates simulation traces from instrumented binaries.

To generate traces properly, we built the Trace Generator using Pin from Intel, a binary instrumentation and analysis tool. Pin allows the development of Pintools, which are programs developed, making use of Pin routines to define which code sections will be analyzed and which kind of analysis or operation must be executed in these sections. These Pintools are then executed with the binary file and perform the analysis during the execution using a just-in-time compiler.

[1] `https://github.com/AlineS/intrinsics`

Inside the Trace Generator, our Pin-tool identifies every Intrinsics-VIMA function call in the binary file. It replaces all x86 assembly code generated during compilation by assembly code correspondent to VIMA. These replacements are written in the output simulation traces, as illustrated in Figure 2.6. Despite the changes inserted in the simulation traces, the behavior and main features defined during the x86 compilation, such as jump addresses and register usage, remain the same.

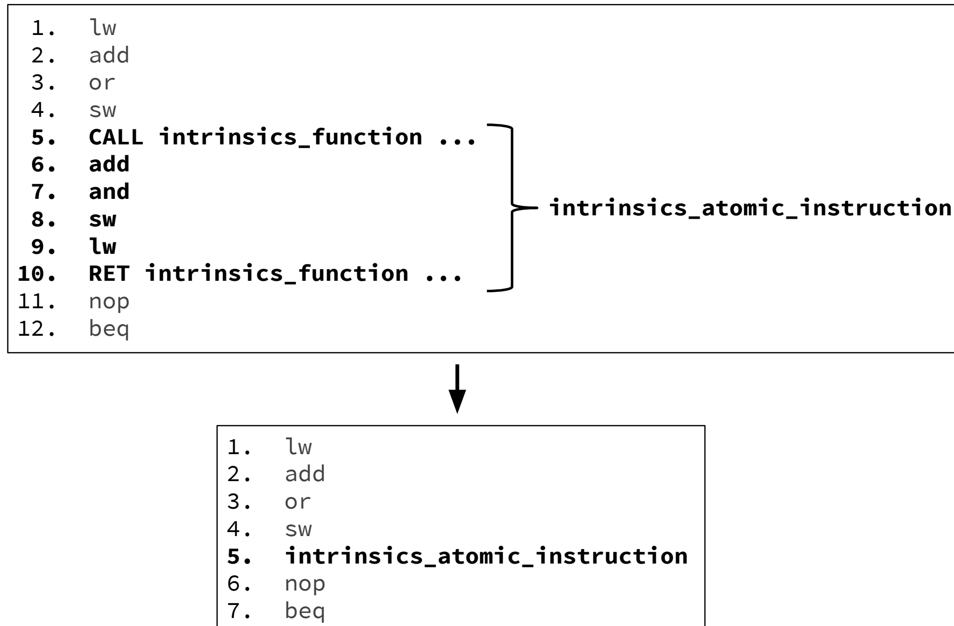


Figure 2.6: Example of x86 assembly replacement. This simplified example shows that the number of instructions is reduced in simulation traces, since blocks of x86 assembly code are replaced for single synthetic VIMA instructions.

Equal to SiNUCA traces, our simulation traces are divided into three types: static, dynamic, and memory. For further details on such traces, please refer to the SiNUCA paper (Alves et al., 2015; Alves, 2014).

When developing non-existing ISA, researchers usually write their simulation traces by hand. However, this is a painstaking and error-prone task, given the complexity from assembly development to correctly placing registers' dependency. Fortunately, OrCS enables users to automatically generate these traces for both existing and non-existing ISAs, thanks to our NDP-intrinsics library.

3 RELATED WORK USING SYSTEMATIC MAPPING

As soon as we defined the topic to be explored and developed in this dissertation, it was necessary to understand the current advances related to architecture and ML areas of research. Being aware of the most remarkable works and studies related to these ideas is also necessary to better understand the field and avoid mistakes such as improper tools, methods, or misconceptions cleared by past studies. Petersen et al. (2015) proposed a technique to create a reliable base for any research work called Systematic Mapping. This technique proposes a way to overview a research area by searching what topics are covered and where this literature has been published. These criteria can bring updates about recent and compelling research topics in areas of interest.

In this chapter, we used the Systematic Mapping methodology to find and understand the state-of-the-art about NDP architectures for ML applications.

3.1 SYSTEMATIC MAPPING METHODOLOGY

Following the Systematic Mapping approach (Petersen et al., 2015), we defined some research questions based on the requirements in porting ML applications to NDP architectures:

- RQ1: What were the main objectives in evaluate ML applications in NDP or PIM architectures?
- RQ2: What kind of hardware was used?
- RQ3: Which components were used to build it?
- RQ4: What were the methods, tools, and mechanisms used to prototype and evaluate?
- RQ5: Which ML applications were chosen?
- RQ6: Which programming languages were used?
- RQ7: Do the researchers presented performance results?
- RQ8: In what institutions were the works developed?

It is possible to delimit this work with those questions. Firstly, by distinguishing processing elements, FUs, and its building blocks. Secondly, by distinguishing the tools used in porting and simulating these work, the programming languages used, and the results obtained. We also found it essential to know where these research works were developed to identify the leading research groups working in this area.

Thereby, we searched papers focusing on ML applications in NDP and PIM architectures in the IEEE Xplore, ACM Digital Library, and Springer Link bases, which provide advanced tools for a more accurate filter of papers. So, a boolean search was used on these bases, filtering by the computer science field.

Thus, we combined the strings "Processing-in-memory", "Processing-in-memory architecture", "In-memory processing", "In-memory architecture", "Near-data processing", "Processing-near-memory", "Data-centric computing", "Near-data computation", "In-memory computing", "Machine learning", "Deep Learning", and "Neural Network" to perform the searches and are presented in Table 3.1.

Table 3.1: Strings used in search base.

Search base	Date	Search string
IEEE Xplore	18/07/2019	(("Processing-in-Memory" OR "In-Memory Processing" OR "Processing-in-Memory architecture" OR "In-Memory architecture" OR "Near-data Processing" OR "data- centric computing" OR "processing-near-memory" OR "in-memory computing") AND ("Machine Learning" OR "Artificial Intelligence" OR "Deep Learning" OR "Neural Network"))
ACM Digital Library	18/07/2019	(("Processing-in-Memory" OR "In-Memory Processing" OR "Processing-in-Memory architecture" OR "In-Memory architecture" OR "Near-data Processing" OR "data-centric computing" OR "processing-near-memory" OR "Near-Data Computation" OR "in-memory computing") AND ("Machine Learning" OR "Artificial Intelligence" OR "Deep Learning" OR "Neural Network"))
Springer Link	18/07/2019	(("Processing-in-Memory" OR "In-Memory Processing" OR "Processing-in-Memory architecture" OR "In-Memory architecture" OR "Near-data Processing" OR "data- centric computing") AND ("Machine Learning" OR "Artificial Intelligence" OR "Deep Learning" OR "Neural Network"))

Table 3.2: Inclusion/Exclusion criteria.

Inclusion criteria:
• Researches focusing on executing ML applications in PIM and NDP architectures or accelerators.
• Researches present performance results.
• Researches using only digital circuits.
Exclusion criteria:
• Researches about ML applications without focusing on architectural aspects.
• Researches about PIM and NDP without focusing on ML applications.
• Researches using analog circuits, such as memristor, Resistive Random-Access Memory (RRAM), Magnetoresistive Random Access Memory (MRAM), Static Random-Access Memory (SRAM), and neuromorphic architectures.
• Surveys only analyzing the evolution of ML or memory and processor.
• Papers are not written in the English language.
• Incremental researches: publications that keep most content of each other just changing or adding some details.

These searches returned 437 papers considering the three search bases above where: 148 were from the IEEE Xplore base, 64 from ACM Digital Library, and 225 from Springer Link, in articles and conference papers. Authors of 57% of these papers were established in the USA or China, and 81% of the papers are 5-year or less. However, we also considered older papers to find relevant information about PIM and NDP concepts and optimized algorithm implementations.

In the first search, we found a large number of papers, so it was necessary to refine it. Thus, we created some inclusion/exclusion criteria, which are presented in Table 3.2. After this step, we selected 27 papers plus one new find from the Google Scholar indexing site. The selected papers are presented in Appendix B.1. Moreover, 25 papers have similar proposes compared to this work, but were excluded according to Table 3.2 by its non-volatile memory and analog circuits employee, such as RRAM, MRAM, and neuromorphic architectures (Chi et al., 2016; Long et al., 2018; Angizi et al., 2018; Cheng et al., 2017; Pan et al., 2018b; Bojnordi and Ipek, 2016; Li et al., 2018; Shafiee et al., 2016; Ji et al., 2016; Agrawal et al., 2018; Lue et al., 2018; Fan and Angizi, 2017; Srivastava et al., 2018; Gupta et al., 2018; Yu et al., 2015; Chen et al., 2018; Kaplan et al., 2018; Cheng et al., 2018; Pan et al., 2018a; Gupta et al., 2019; Salamat et al., 2018; Imani et al., 2019b,a; Jiang et al., 2019; Imani et al., 2018) since these works use different processing approaches and technologies, such as current and voltage variations to process analog signals to calculate a NN.

Between those 27 papers, we selected only one from the 19% that referred to the older papers. Generally, these papers' main goal is to achieve better performance and less energy consumption by executing ML applications. From these works, 59% of the papers explicitly consider NDP or PIM concept; 82% presents parallelism and code optimization; 56% developed frameworks to implement ML codes; 6% consider different architectures such as Very Long Instruction Word and Automata Processor. Initially, we did not consider specific-purpose hardware research since this work employs a general-purpose processor. However, the amount of work using specific-purpose processors was considerable, and some of these papers are relevant to this work. Therefore, after following the guidelines of systematic mapping, we considered 18 papers as related work: Cadambi et al. (2010), Thottethodi et al. (2018), Li et al. (2017), Ahn et al. (2016), Xu et al. (2015), Gao et al. (2015), Oliveira et al. (2017a), Azarkhish et al. (2018), Gao et al. (2017), Gao et al. (2018), Schuiki et al. (2018), Liu et al. (2018), Min et al. (2019),

Table 3.3: Summary of correlated papers characteristics.

Paper	General / Specific Purpose	Vector / Scalar	Near-memory / In-memory	# Full cores
Cadambi et al. (2010)	General	Vector	Near-memory	N
Thottethodi et al. (2018)	General	Vector	Near-memory	N
Li et al. (2017)	General	Vector	In-memory	1
Ahn et al. (2016)	General	Scalar	Near-memory	N
Xu et al. (2015)	General	Scalar	Near-memory	N
Gao et al. (2015)	General	Scalar	Near-memory	N
Oliveira et al. (2017a)	General	Vector	In-memory	1
de Lima et al. (2019)	General	Vector	Near-memory	N
Azarkhish et al. (2018)	Specific	Vector	Near-memory	N
Gao et al. (2017)	Specific	Vector	Near-memory	N
Gao et al. (2018)	Specific	Scalar	In-memory	1
Schuiki et al. (2018)	Specific	Scalar	Near-memory	N
Liu et al. (2018)	Specific	Scalar	Near-memory	N
Min et al. (2019)	Specific	Scalar	Near-memory	1
Deng et al. (2018)	Specific	Scalar	In-memory	1
Ganguly et al. (2018)	Specific	Scalar	Near-Memory	1
Sim et al. (2018)	Specific	Scalar	In-memory	1
Deng et al. (2019)	Specific	Vector	In-memory	1
VIMA	**General**	**Vector**	**In-memory**	**1**

Deng et al. (2018), Ganguly et al. (2018), Sim et al. (2018), de Lima et al. (2019), and Deng et al. (2019) and they were summarized in Table 3.3, which highlights four characteristics considered important by us considering migration of ML applications to a NDP architecture and will be explained below.

To understand how these previous work are related and to point the similarities and differences between this work and the previous ones, we selected some relevant characteristics relying on architecture to achieve high performance for ML algorithms, such as:

(i) **A general or specific-purpose architecture:** Focus on a general-purpose architecture that can achieve high performance in ML applications without disregarding another kind of application.

(ii) **Vectorial or scalar operations:** To consider vectorization of all Reduced Instruction Set Computer (RISC) operations used by ML applications, such as multiplication, addition, and boolean instructions.

(iii) **Using near-memory or in-memory approach:** Focus on near-memory approach, in other words, making an effort to integrate a 3D-memory instead of trying to change memory cells to perform calculations.

(iv) **Integrating full cores or develop a simple circuit to perform operations:** Full cores normally are more expensive computationally and in energy costs than simplified circuits or FUs to be attached to the 3D-memory. These characteristics can discriminate VIMA from others.

3.2 STATE-OF-THE-ART

After reading the related work, we summarized information about the architecture configurations in Table 3.3. Liu et al. (2018), Cadambi et al. (2010), Thottethodi et al. (2018), Li et al. (2017), Sim et al. (2018), Deng et al. (2018), Azarkhish et al. (2018), Schuiki et al. (2018), Min et al. (2019), Ganguly et al. (2018), Xu et al. (2015), Gao et al. (2015), Deng et al. (2019), Sim et al. (2018), de Lima et al. (2019), and Sudarshan et al. (2019) rely on simulators to evaluate computational performance. Some of them also developed an Application Programming Interface (API) to provide a way for programmers to implement codes for their PIM and NDP architectures.

Liu et al. (2018), Cadambi et al. (2010), Ahn et al. (2016), Gao et al. (2018), and Sim et al. (2018) developed APIs to enable programmers to use their architectures in an easy way, with high-level code. For example, the API developed by Sim et al. (2018) allows the programmer to configure the system parameters and to allocate DRAM regions to execute specific functions. While Gao et al. (2015) and Gao et al. (2018) developed low-level APIs, in which the programmer's code remains the same, but the API initializes the NDP architecture, set the system environment configurations, executes synchronization, communication and memory mapping. Besides, the API developed by Ahn et al. (2016) implements different function calls to allow parallel programming considering their specific cores. Cadambi et al. (2010) also implemented a set of functions to be used in high-level code that can map Processing Elements (PEs) in memory and distribute data and functions between them. Liu et al. (2018) proposed an OpenCL extension to profile applications and dynamically map and schedule them into the architecture cores.

In contrast, VIMA is very simple as the programmer needs only to write a C/C++ code using Intrinsics-VIMA without worrying about the execution or memory allocation inside the architecture. Furthermore, our model is based on an x86 general-purpose architecture. The processor will fetch every instruction and send it to VIMA only when treating NDP instructions. The rest of the instructions are executed by the x86 processor using its FUs.

Gao et al. (2015) approach is similar to VIMA considering the API usage and simulation. Besides, it uses an API for the programmer to write the application without worrying about the execution inside the PIM architecture. Gao et al. (2018), Li et al. (2017), and Schuiki et al. (2018) use a library to compile the application and their scheduling code is done in low-level. Meanwhile, Gao et al. (2018), and de Lima et al. (2019) use a linked library to generate PIM application in x86 environment to be simulated in Gem5 simulator. Finally, Liu et al. (2018), and Ahn et al. (2016) also use Intel Pin to generate traces to simulate them. Most of these consider architectures with many cores inside the memory, so their API also enables the programmer to allocate the cores and distribute the functions to them, configure the accelerator, update page tables, and call their library functions. Besides, some consider full cores or simple cores for each vault, also allowing communication between the cores. VIMA's API and architecture are simpler than related work. Intrinsics-VIMA library shall be included and called in a conventional C/C++ implementation. While the VIMA architecture mainly requires a sequencer, a module of FUs, and cache memory to compute data.

We can divide related work into two groups, those based on 3D-stacked memories and those relying on Dual Inline Memory Module (DIMM) to accelerate ML applications. Thus, focusing on criteria described in Table 3.3, 12 related work used the 3D-stacked memories: Liu et al. (2018), Ahn et al. (2016), Thottethodi et al. (2018), Gao et al. (2015), Oliveira et al. (2017a), Hong et al. (2018), Azarkhish et al. (2018), Schuiki et al. (2018), Min et al. (2019), Gao et al. (2017), Xu et al. (2015), de Lima et al. (2019) while 6 modified conventional DRAM memories with integrated circuits: Deng et al. (2018), Gao et al. (2018), Li et al. (2017), Deng et al. (2019), Sudarshan et al. (2019), Sim et al. (2018). Among the related work using 3D-stacked memories

Azarkhish et al. (2018), Schuiki et al. (2018), Liu et al. (2018), Ahn et al. (2016), Gao et al. (2015), and Xu et al. (2015) employ full processing cores such as RISC-V, ARM, and Accelerated Processing Unit (APU), and Gao et al. (2017), Min et al. (2019), Thottethodi et al. (2018), and Cadambi et al. (2010) developed specific-purpose embed cores.

3.2.1 NDP Approaches with Full Cores

Considering the related work that used 3D-stacked memories, we could observe that some integrated full processing cores or simple circuits in it. NeuroStream (Azarkhish et al., 2018) is a scalable PIM platform capable of running DNNs with large input sizes and arbitrary filter sizes, and based on NeuroStream, Schuiki et al. (2018) implements a near-memory acceleration engine that can be used to train state-of-the-art Deep Convolutional Neural Networks.

Both proposals implement a module composed of RISC-V cores with local cache, Direct Memory Access (DMA), and specific cores. The first work is called NeuroStream and the second one, Network Training Accelerator. These modules are connected to every 3D-stacked memory using the crossbar switch, and both proposals enable vectorizing a few instructions. Except for the use of several cores in these previous proposals, this dissertation is similar if considering that it attaches the VIMA vector module to the crossbar switch. However, considering that this VIMA module needs only a few components (a cache, FUs, and a sequencer), it is clear the simplicity of our proposal.

A different proposal by Liu et al. (2018) developed a heterogeneous PIM architecture to accelerate ML training models developed in conventional frameworks. This proposal implements in the HMC logic layer, a series of fixed-function logic and programmable cores. The fixed-function cores are composed of adders and multipliers, and the programmable functions are ARM-based cores, which can be loaded and offloaded with kernel functions through a communication scheme with the CPU host.

The Tesseract approach (Ahn et al., 2016) focuses on accelerating large-scale graph processing using an HMC module and integrating what is called a Tesseract core, a single-issue in-order ARM core. These cores can communicate with each other by a message-passing protocol because each core holds a local memory. Additionally, it uses two prefetch schemes to exploit the memories better. Although this work focuses only on graphs, it can be considered a related work, as graphs can represent NNs.

The NDP architecture proposal by Gao et al. (2015) develops the hardware and software of an NDP architecture for in-memory analytics frameworks, including MapReduce, graph processing, and DNNs. The authors employed a set of NDP ARM cores, Translation Lookaside Buffers (TLBs) and, virtual memory schemes. These cores can communicate through a vault router and share the physical address with the CPU host.

The fact of using a set of ARM and RISC-V cores or APUs in a 3D-staked memory logic layer makes those solutions too expensive computationally compared to VIMA. As mentioned above, our proposal relies on a PIM-enabled memory with a vector module attached to the crossbar switch composed of a cache, vector FUs, and a sequencer. Besides, VIMA does not rely on communication between vaults, sharing of physical address space, or complex virtual memory schemes to achieve high performance or parallelism. Instead, the host CPU communicates with the PIM-enabled memory during execution, identifying VIMA instructions, and sending it to the NDP device.

3.2.2 NDP Approaches with General-Purpose Cores

Still considering the related work relying on 3D-stacked layers, some are general-purpose architectures that implement simple circuits, such as Neuron In-Memory (NIM) (Oliveira et al., 2017a), a PIM reconfigurable accelerator that can simulate biologically meaningful NNs of considerable size. NIM is a module composed of a register bank, complex processing units, and a sequencer. It is attached to the crossbar switch, with one NIM module for each vault. Although NIM enables executing vector instructions, its architecture has a design more complicated and expensive than VIMA due to requiring a module for each vault. In contrast, VIMA has only one module attached to the crossbar switch, enabling communication to all vaults.

Xu et al. (2015) focuses on the parallelization of CNN on a system with multiple PIM devices. The authors consider two APUs, which consists of CPU and GPU cores. One APU as a host connected to a 3D-memory and the other APU is integrated into the memory's logic layer. In this way, their module has additional overhead due to the bus communication or syscalls to use these devices, being less energy efficient than VIMA.

Finally, de Lima et al. (2019) considers a reconfigurable mechanism inside the PIM module to dynamically reduce or increase the number of active FUs as the application demands. To do so, they considered a Reconfigurable Vector Unit (RVU) module to each vault, which comprises a set of 32×8-byte multi-precision FUs, a Finite State Machine (FSM) to control the flow of RVU instructions and an 8×256-byte register file. The RVU modules can operate independently and execute vector instructions. Similar to our proposal, they use a specific compiler to generate simulation traces to the Gem5 simulator. However, their architecture has a design more complex than ours since it dynamically adjusts itself during execution. Besides, they still need a RVU for each vault to vectorize the execution.

3.2.3 NDP Approaches with Embedding Specific-Purpose Cores

Among the related work relying on 3D-stacked layers, some implement simple and specific circuits, such as TETRIS (Gao et al., 2017) that is a proposal of a PIM NN accelerator, a software scheduling, and partitioning techniques. For each vault in a 3D-stacked memory, the authors propose hundreds of PEs connected through a dedicated network. Each PE is composed of an Arithmetic Logic Unit (ALU), an SRAM, and a register file. All PEs share a global buffer for communication.

NeuralHMC (Min et al., 2019) is a proposal of an HMC-based accelerator tailored for efficient DNN execution. NeuralHMC is based on communication between HMCs vaults to achieve higher parallelism, so they implement HMCs communication with Network-on-Chip (NoC), such as TETRIS. Just considering the communication between vaults to allow greater parallelism makes both proposals more expensive than VIMA as VIMA keeps the HMC packet communication (Hybrid Memory Cube Consortium, 2014), which is a simpler solution.

The MAssively Parallel Learning/Classification Engine (MAPLE) proposal Cadambi et al. (2010) uses a parallel accelerator for learning and classification applications and a tool to map application kernels to the accelerator hardware automatically. The authors of MAPLE implemented an architecture with a set of cores to solve MapReduce operations. These cores are composed of processing elements that cover registers, selectors, a vector FU, and a local store. This set of PEs handles intermediate data, which is solved by a Smart Memory Block, capable of atomic store operations. They implement parallel concepts in hardware, such as intermediate operators. Compared to VIMA they need a significant amount of processing cores to achieve parallelism and two different modules to solve the entire operation, while VIMA needs a single and simple module that enables vector operations by default.

The Millipede proposal (Thottethodi et al., 2018) uses a processing near-memory architecture for Big data Machine Learning Analytics. Millipede processors are attached to a 3D-stacked memory's logic layer and are composed of corelets, local memory, a register file, a pipeline, and prefetch buffers. Each corelet has its instruction cache. Millipede employs Multiple Instruction, Multiple Data (MIMD) operations and uses prefetch to share data between corelets. Even enabling MIMD execution, the high number of cores used in their work and the massive prefetch communication are too expensive and with a more complex design compared to VIMA.

Ganguly et al. (2018) analyzes the aspects of a CNN algorithm to develop a NDP architecture capable of achieving high computational performance for this specific algorithm. In their work, the authors used cores in the HMC logic layer, so the data controller from each vault passes data through the multiplication and then through the sum units. This hardware is formed by multiple layers of neurons to parallelize CNN operation. Besides, the vaults can communicate by using the data controller. Although VIMA's FU may be more expensive than these addition and multiplication hardware, since it allows the execution of integer and FP instructions, it is general-purpose hardware. It allows the execution of any application written with Intrinsics-VIMA, not just CNNs.

3.2.4 DRAM and PIM Approaches

Among the related work that relies on flat DRAM memories, Gao et al. (2018) implements an accelerator system in hardware with a software interface for support. This accelerator is composed of near-DRAM control logic and a few computational kernels implemented by pre-stored Look-Up-Tables (LUTs) implemented into the DRAM. Each LUT represents a basic unit for different critical operations and packets make the communication between host and accelerator.

DrAcc (Deng et al., 2018) and DRISA (Li et al., 2017) are two similar approaches, as both proposals implement an accelerator built with DRAM technology. DrAcc focuses on Ternary Weight Neural Networks and implements a carry look-ahead adder inside DRAM memory. DRISA is a reconfigurable architecture to achieve high parallelism by operating data inside DRAM cells. Both proposals implement boolean circuits inside DRAM memory.

LAcc (Deng et al., 2019) is a proposal of a DRAM-based PIM accelerator that supports LUT based fast and accurate vector multiplication for CNNs. LAcc uses a multiplication decomposition to achieve acceleration. This decomposition allows splitting an operand to multiply 2-bits at a time and then group it in LUTs inside the DRAM. For each DRAM bank, an LUT is implemented to operate vectors of weights.

NID (Sim et al., 2018) is a proposal to perform a binary convolution efficiently by exploiting in-DRAM bulk bitwise operations. The authors of the proposal implement logic in DRAM memory and allocate inputs and kernels to DRAM banks to optimize performance. The kernels are split into multiple parts to execute partial computations. Max Pooling, normalization, and activation layers are processed out of memory since their computational complexity is extremely low, so additional digital blocks are implemented in the peripheral area of a DRAM.

NNDRAM (Sudarshan et al., 2019) is a proposal of a DRAM-based in-memory Binary Weight Neural Network architecture to minimize the energy and exploit maximum data parallelism. This previous proposal aims to integrate a neural network basic computation unit next to the sense amplifiers.

All these mentioned work above rely on adding a boolean circuit to DRAM cells, which is not an expensive task. However, compared to VIMA these previous work require a more complicated algorithm implementation.

3.2.5 Conclusions on Related Work

In short, compared to VIMA, the related work present similar ideas but execute it using different methods, as listed below:

- Specific-purpose hardware is an effective approach since it achieves better computational performance to execute the application. However, it is not flexible enough to solve general-purpose applications neither to keep a high computational performance during its execution.
- Solutions implemented inside the DRAM can consume low energy and have low computational complexity. However, they require much effort from the designers, as enabling DRAM cells to calculate a restricted set of instructions is non-trivial from an electrical engineering and manufacturing perspective.
- Using a large number of cores to achieve higher parallelism can introduce much more computational complexity to the hardware and, consequently, higher energy cost.

Considering that VIMA relies on the opposite of these characteristics, we expect it to achieve promising results using the full parallelism present on 3D-stacked memories, also allowing near-data reuse, thanks for its cache. Besides, the Intrinsics-VIMA library will enable multiple instructions that are not restricted just to ML applications.

4 MACHINE LEARNING CODE PORTABILITY

In this chapter, we detail the Intrinsics-VIMA library, which we developed to evaluate ML applications performance in VIMA, a vector NDP architecture. Intrinsics-VIMA is a library created intending to facilitate the development of programs for NDP architectures using the *C/C++* language. We used it to port three kernels of applications widely adopted in ML to this new architecture, kNN, MLP, and CNN. These applications are also described in this chapter, detailing the method to vectorize each of them.

We used the ported ML applications binary files to generate simulation traces with the Trace Generator. Finally, the trace was simulated by OrCS in order to obtain memory and processor behavior. The main idea for this dissertation was to evaluate the performance of these applications using Intrinsics-VIMA compared to their implementation with AVX instructions, and the results are shown in Chapter 5. Figure 4.1 illustrates the workflow from the steps described above.

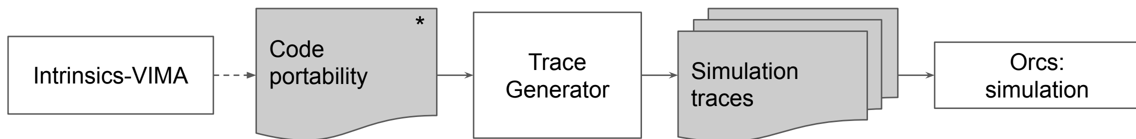


Figure 4.1: Sequence of steps to simulate an application.

4.1 INTRINSICS-VIMA

This library is based on vector instructions from ARM NEON Intrinsics and RISC instructions from Microprocessor without Interlocked Pipeline Stages (MIPS) ISAs. It implements simple arithmetic, logic, and comparison instructions, listed in Appendix A.1. It also supports trace generation for simulation and enables vector operations with vectors of 256 B and 8 KB formed by multiple integers, single-precision or double-precision FP elements. Thus, depending on variable representation and vector size, a function can operate over 2048 × 32-bits elements, 1024 × 64-bits elements, 64 × 32-bits elements, or 32 × 64-bits elements.

The main idea for Intrinsics-VIMA is to provide vector extensions in the x86 ISA. By implementing *C/C++* code using Intrinsics-VIMA, it can be debugged and executed on any architecture. However, to evaluate new NDP architectures, our trace generator is necessary to transform each intrinsic call into a specific NDP instruction supported by the simulator.

Algorithm 4.1 presents the implementation of a vector sum example using Intrinsics-VIMA. Algorithm 4.2 shows the implementation of one of our Intrinsics-VIMA routine.

Algorithm 4.1: Intrinsics-VIMA routine call for vector sum.

```
1  uint32_t vima_size = 2048;
2
3  // Allocate the vectors A, B (sources) and C (result)
4  __v32f *A = (__v32f*)malloc(sizeof(__v32f) * vima_size * x);
5  __v32f *B = (__v32f*)malloc(sizeof(__v32f) * vima_size * x);
6  __v32f *C = (__v32f*)malloc(sizeof(__v32f) * vima_size * x);
7
8  // Initialize the memory location
9  <...>
10
11 // Perform the vector sum: C[i] = A[i] + B[i]
```

```
12  for (int i = 0; i < vima_size * x; i += vima_size) {
13      _vim2K_fadds(&A[i], &B[i], &C[i]);
14  }
```

Algorithm 4.2: Intrinsics-VIMA routine example.

```
1  // This routine can be fully executed in any architecture
2  // Our simulator replaces this routine with a VIMA instr.
3  void *_vim2K_fadds(__v32f *a, __v32f *b, __v32f *c) {
4      for (int i = 0; i < vima_size; ++i) {
5          c[i] = a[i] + b[i];
6      }
7      return EXIT_SUCCESS;
8  }
```

The nomenclature of VIMA instructions follows the rules described below:

1. The first character (a single *underline*) indicates a function;
2. The next three characters indicate the current architecture, in this case, VIMA;
3. The next two characters indicate the number of elements in a vector that can be executed by the function;
4. The next two characters (initiated by an *underline*) indicate the x86 operator data type (integer or floating-point);
5. The next three characters indicate which operation to be executed; and
6. The last character indicates signed or unsigned operands.

For each combination of variable representation and vector size, there is an option for signed or unsigned variables, so the data types help in organizing all the parameters, as illustrated in Table 4.1.

Table 4.1: Intrinsics-VIMA data types.

Data type	**Description**
__VM64I	4-bytes value in a vector of 64 positions
__VM2KI	4-bytes value in a vector of 2048 positions
__VM32L	8-bytes value in a vector of 32 positions
__VM1KL	8-bytes value in a vector of 1024 positions

The nomenclature for the data types follows the rules below:

1. The first two characters initiate with double *underline* and indicates a data type;
2. The next two characters indicate the current architecture, in this case, VIMA;
3. The next two characters indicate the number of positions in a vector;
4. The last character indicates the operator x86 size (integer or 32 b and long or 64 b).

The main idea for this ISA is to develop the simplest vector operations which can be combined to solve more complex operations. Additionally, simple instructions are implemented in different ISAs with known execution costs, so it is easy for us to estimate VIMA functions' costs in the simulator. As mentioned in Chapter 2, all Intrinsics libraries and some usage examples are available in our GitHub [1].

[1] https://github.com/AlineS/intrinsics

4.2 OVERVIEW: MACHINE LEARNING KERNELS

This section explains the three algorithms used in this dissertation, providing necessary details and an overview before their migration to our NDP architecture.

4.2.1 Convolution Basics

Convolution operations are based on repeatedly updated values in a matrix by combining the neighbors of a point, typically using the sum of products. It has applications that include statistics, probability, computer vision, natural language, image processing, computational electromagnetic, and differential equations (Krishnamoorthy et al., 2007; Lacassagne et al., 2014; Holewinski et al., 2012)

A convolution computes values based on a fixed pattern involving each element of an array and several neighbors on a two- or three-dimensional arrangement (Afonso et al., 2017). Considering 2D operations, two of the most common convolution patterns are the Von Neumann and Moore neighboring. The first pattern includes the four neighbors in the cardinal directions of an element (Toffoli and Margolus, 1987) and is illustrated in Figure 4.2. The second pattern includes all the eight neighbors around a cell (Zaitsev, 2017), as depicted in Figure 4.3.

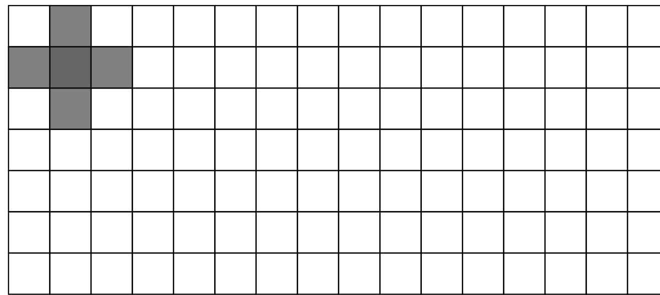


Figure 4.2: Von Neumann neighboring convolution.

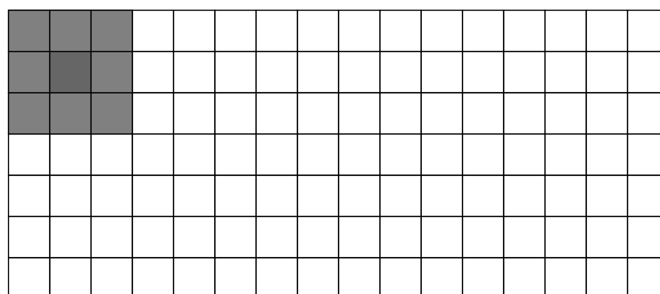


Figure 4.3: Moore neighboring convolution.

The computation of each element is independent, making convolution codes good candidates for parallel processing. However, they often become memory bottlenecks due to the data access patterns they present potentially having poor locality (Afonso et al., 2017).

For ML, convolution works as a feature extractor for images, where it preserves the spatial relationship between pixels and the main characteristics of the image (Lawrence et al., 1997). Different filters are applied to extract these features in each input image by calculating the convolution. Each filter is a tiny square matrix and represents a specific feature. For this work, we implemented a Moore neighboring convolution.

4.2.2 k-Nearest Neighbors (KNN) Basics

kNN is one of the simplest instance-based classifiers. It searches for the k minimal distances between training and test points in an n-dimensional space. kNN uses the Euclidean distance method to calculate those distances. Each instance is represented by a feature vector, which is an n-dimensional array of features. In other words, each array position corresponds to a different feature, tied to a weight (Mitchell, 1997).

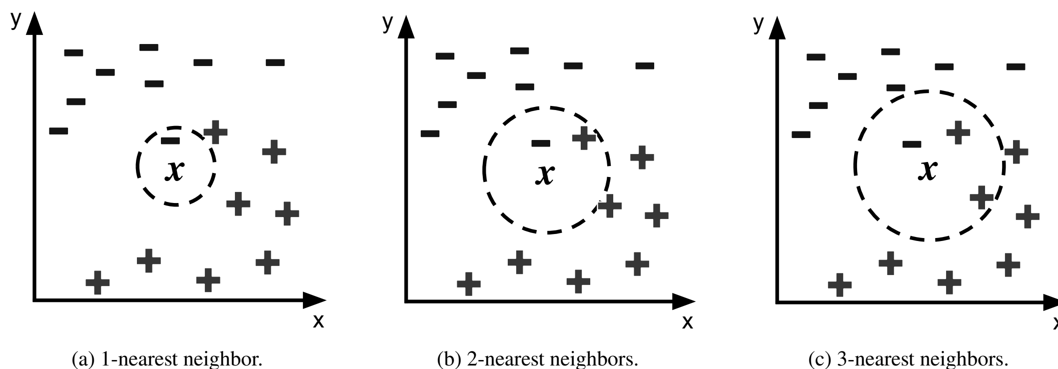
For example, given an instance x, it is represented by the feature vector (below), where $a_r(x)$ denotes the value of the *rth* attribute of instance x.

$$\{a_1(x), a_2(x), \ldots, a_n(x)\}$$

So, the distance between two instances x_i and x_j is defined by $d(x_i, x_j)$, as in the equation below:

$$d(x_i, x_j) \equiv \sqrt{\sum_{r=1}^{n}(a_r(x_i) - a_r(x_j))^2}$$

In this manner, the kNN algorithm stores the whole training database during execution then calculates Euclidean distance between each test to each training instance in the database. The k smallest distances are selected to classify the test instance by vote. In other words, these k training instances are the closest points to test instance point in Euclidean space, so the most representative label value between these k instances indicate the test instance class as illustrated in Figure 4.4.



(a) 1-nearest neighbor. (b) 2-nearest neighbors. (c) 3-nearest neighbors.

Figure 4.4: Instance classification by votes. The figures show how the voting scheme works in order to classify the test instance x in positive or negative classes. In Figure 4.4(a), $k = 1$, so only the closest training point can classify x. In this case, x is labeled as belonging to the negative class. Figure 4.4(b) is an example explaining why k can not be an even number due to tied votes. In Figure 4.4(c), x is classified as positive due to the higher amount of positive neighbors, showing that k can affect classification and the researcher must understand the database in which he or she is working, in order to choose good parameters.

4.2.3 Multi-Layer Perceptron (MLP) Basics

The MLP algorithm is a computational model that is inspired by biological neurons and the human brain. It is advantageous to solve stochastic problems since it computes approximation functions. Thus, one of the uses of the MLP application is to create mathematical models with data samples to identify relationship patterns between features and targets (instance label) and finally classify other samples.

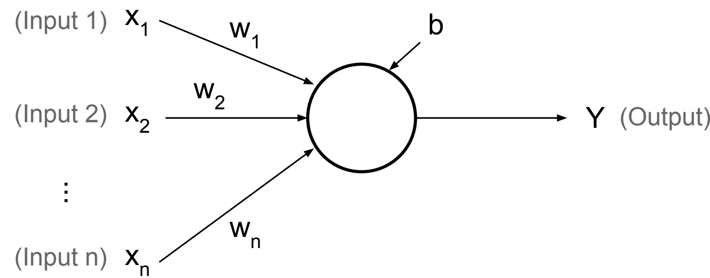


Figure 4.5: A neuron representation.

A neuron, also called perceptron, is the basic unit in a NN and is illustrated in Figure 4.5. Each neuron calculates an output value **Y** with an activation function f using the input values it receives as parameters, as the following equation:

$$Y = f(w_1.x_1 + w_2.x_2 + \ldots + w_n.x_n + b)$$

In the equation, Y is the neuron that represents an activation value. The other inputs refer to neurons' activation value from the previous layer (x_n) and the weights of connections between neurons (w_n). These values are multiplied, accumulated, and added to the bias value (b), which is a correction value for the activation function, helps to achieve higher accuracy. The activation function applies non-linearity into the neuron output during NN training and testing.

This organization leads to the concept of fully connected layers, as illustrated in Figure 4.6. In this example, multiple layers of neurons are connected, where each neuron has an activation value, a bias value, and weighted connections. The NN is composed of at least 3 different layers:

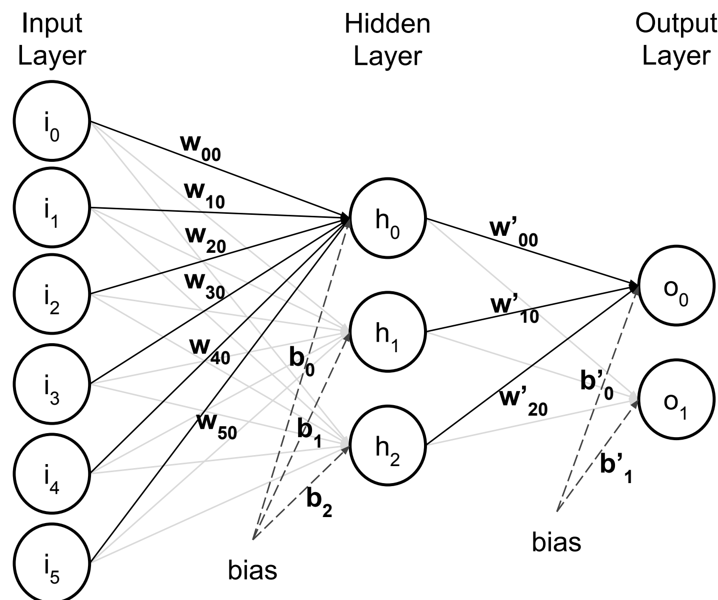


Figure 4.6: A representation of a neural network.

- **Input Layer:** represents the neurons with outside information. It is represented by an array, so each position is a neuron with an activation value.

- **Hidden Layer:** an intermediary layer that transfers and adjust input information to NN output with the activation function. There may be one or multiple hidden layers in a NN. **Output Layer:** depends on the number of classes the problem present. Like the hidden layer, depending on the nature of the data, it uses an appropriate activation function to transform the output activation values (Bishop, 2006). The final activation values indicate which is the most relevant classification or feature for a specific instance.

As illustrated in Figure 4.6, the computation starts at the input layer in order to calculate the activation values from the hidden layer. Considering that the input layer has *n* neurons and the hidden layer has *m* neurons, then each input neuron must have *m* weighted connections. When the activation values corresponding to the hidden layer were all calculated, the same operations are repeated to calculate the output layer's activation values. This process, from input to the output layer, is called Forward Propagation. Depending on the algorithm and the dataset, a specific objective function is applied in this step to highlight the most representative features and results present in the analyzed dataset.

However, at the beginning of the training, the NN will not correctly classify instances because its values and parameters may not be adjusted to be accurate enough, resulting in low and similar distributions among the output neurons, as illustrated in Figure 4.7. Then it is necessary to minimize the objective function considering the obtained results and the desired one, to understand how far the NN classification is compared to the expected value. In other words, we have to minimize this classification error. This process is the beginning of the Back Propagation step, which uses this error value to adjust the NN weights. In the next Forward Propagation step, considering the updated parameters, the objective function can obtain distributions that can more closely resemble the expected value.

To minimize a objective function, is normally used algorithms such as Gradient Descent, that find coefficients to minimize a function. These coefficients are applied to adjust the weights to achieve a more accurate classification result in the next Foward Propagation step. During the NN training, the algorithm evaluates a set of training instances in Forwarding and Back Propagation steps, updating the weight's values to improve the classification of a test dataset.

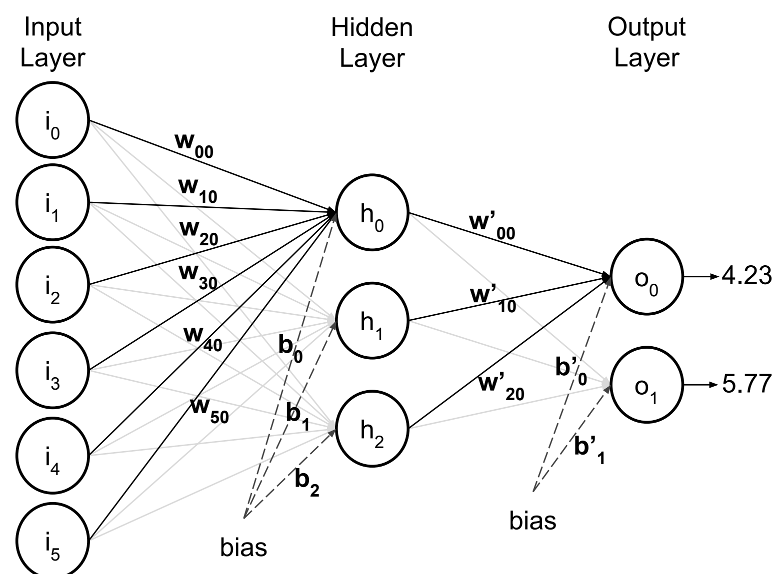


Figure 4.7: Neural network output distribution at the beginning of the training.

As the weight of a neuron connection increases during training, the more relevant it becomes. The NN can evaluate the whole training database multiple times to achieve better

accuracy. Each evaluation of a database is called an epoch. Typically, after each epoch, the weights become more accurate. This process is executed to the whole training database for a specific number of epochs and/or a convergence threshold value (CTV) can be defined by checking the error rate between the current and the previous epochs. Whenever this error is acceptable, it finished training. During this process, the NN shall learn the patterns and achieve good accuracy in the test database.

In this work, we implemented a simple NN for our evaluations. We skipped the training step, so just the inference was represented, with all the training parameters set in constant values. We chose to simplify it because the main achievement here is to show the computational performance of a NN during classification. After all, end-users will almost always only see the inference execution time.

4.3 CODE PORTABILITY: VIMA

Here, we focus on the feasibility of adopting the Intrinsics-VIMA library to develop the three kernels. We implemented all the algorithms with VIMA vectors of 256 B and 8 KB, thus operating over 64 and 2048 single-precision FP values with a single instruction.

The version of the algorithms shown in this work was not the most aggressive, considering the entire vectors' usage. We tried other alternatives before, but this version was chosen due to the clarity in its explanation since it follows a clear pattern. Besides, this version can make better use of VIMA caches, mainly for vectors with 8 KB in size, which makes the cache capable of handling only a few lines, thus needing a careful cache line reuse.

Like HIVE (Alves et al., 2016), although VIMA instructions operates over 256 B and 8 KB vectors, the physical implementation of these architectures can use fewer vector units in a pipeline manner to still provide high performance while low area usage.

4.3.1 Convolution Migration

To implement a naive convolution code using VIMA, we adopted the Moore pattern with a reach equal to 1, as shown in dark gray in Figure 4.3. The algorithm sums all nine elements in the convolution, then multiplies the result by a constant and stores the result in a different matrix. Algorithm 4.3 shows an example in C, considering a matrix in a continuous array arrangement. The algorithm stores the result in the corresponding element of a new matrix.

Algorithm 4.3: Moore convolution code in C.

```
1  for (int i = col_size; i < max_elem; i++) {
2      input2[i] = input1[i];               // Fifth elem.
3      input2[i] += input1[i-col_size-1];   // First elem.
4      input2[i] += input1[i-col_size];     // Second elem.
5      input2[i] += input1[i-col_size+1];   // Third elem.
6      input2[i] += input1[i-1];            // Fourth elem.
7      input2[i] += input1[i+1];            // Sixth elem.
8      input2[i] += input1[i+col_size-1];   // Seventh elem.
9      input2[i] += input1[i+col_size];     // Eighth elem.
10     input2[i] += input1[i+col_size+1];   // Ninth elem.
11     input2[i] *= weight;
12 }
```

Figure 4.8 illustrates the vector convolution. For every loop, unaligned elements from three consecutive lines of the matrix, as pictured in dark gray, are loaded into VIMA vectors and operated over. Algorithm 4.4 shows the implementation using Intrinsics-VIMA. This implementation considers a convolution that eliminates the matrix borders during execution.

Algorithm 4.4: Moore convolution using Intrinsics-VIMA.

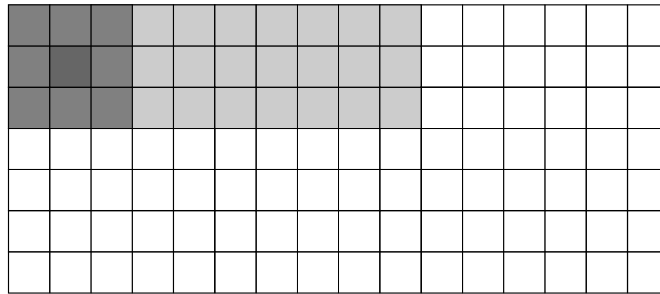


Figure 4.8: Moore neighboring convolution.

```
1  for (int i = col_size; i < max_elem; i += vec_size) {
2      _vim2K_fmovs(&input1[i], &input2[i]);                          // Fifth elem
3      _vim2K_fadds(&input2[i], &input1[i-col_size-1], &input2[i]);   // First elem
4      _vim2K_fadds(&input2[i], &input1[i-col_size], &input2[i]);     // Second elem
5      _vim2K_fadds(&input2[i], &input1[i-col_size+1], &input2[i]);   // Third elem
6      _vim2K_fadds(&input2[i], &input1[i+1], &input2[i]);            // Fourth elem
7      _vim2K_fadds(&input2[i], &input1[i-1], &input2[i]);            // Sixth elem
8      _vim2K_fadds(&input2[i], &input1[i+col_size-1], &input2[i]);   // Seventh elem
9      _vim2K_fadds(&input2[i], &input1[i+col_size], &input2[i]);     // Eight elem
10     _vim2K_fadds(&input2[i], &input1[i+col_size+1], &input2[i]);   // Ninth elem
11     _vim2K_fmuls(&input2[i], &weights[i], &input2[i]);
12 }
```

4.3.2 k-Nearest Neighbors (KNN) Migration

For the kNN algorithm, the training data must be stored in memory so that each test instance can use it for classification, as depicted in Figure 4.9. Depending on the number of features an instance presents, it can be smaller than a VIMA vector, so different instances can be stored consecutively in one VIMA vector as depicted in Figure 4.10. Meanwhile, if the instance's size is equal to or greater than a VIMA vector, it will occupy at least one VIMA vector.

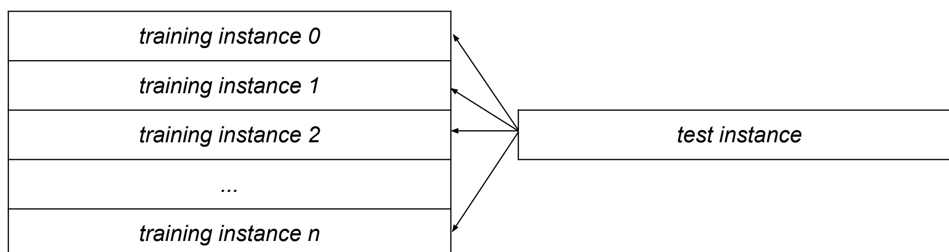


Figure 4.9: The whole training dataset has to be available for each test instance.

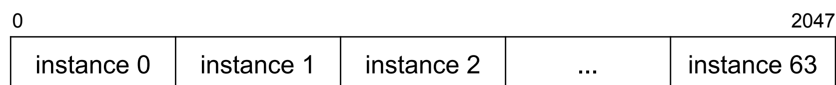


Figure 4.10: Full utilization of a VIMA vector for training and test instances. In this case, it is possible to allocate 64 instances with 32 features inside the vector of 8 KB.

Our training dataset has two classes: 0 (negative) and 1 (positive). The labels are loaded into separated vectors as soon as the training dataset is stored in memory. To do so, a vector with a size multiple of the VIMA vector size must allocate the training labels. Thus, to store a set of 8192 training instances, it will be necessary to use 4 VIMA vectors of 8 KB to store the 8192 labels, as depicted in Figure 4.11.

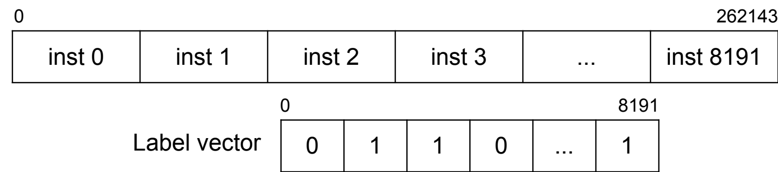


Figure 4.11: VIMA vectors with training instances with 32 features and the respective labels.

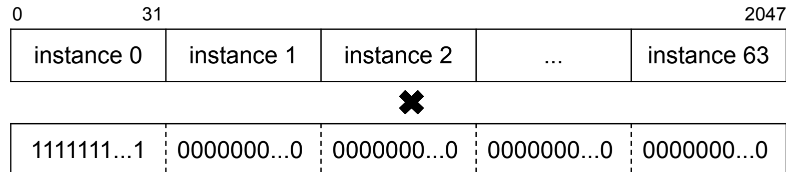


Figure 4.12: Operation to apply a mask over a VIMA vector of 8 KB with instances representing 32 features.

Fortunately, to calculate the Euclidean distance method, which we explained in Section 4.2, most of the operation can be vectorized with Intrinsics-VIMA, such as the following routines:

- *_vim2K_fsubs()* to subtract the values of the training and test instances;
- *_vim2K_fmuls()* to multiply and raise the resulting value to the power of two;
- *_vim2K_fcums()* to sum all partial results to find out the distance between the instances and finally calculates the square root of this value.

Although a VIMA vector can receive more than one instance, depending on the number of features it represents, just a single instance will be computed at a time in this algorithm. For this, a mask is applied in training and test vectors to select just a single instance. For example, considering a VIMA vector of 8 KB and test and training instances with 32 features, the mask will set the first 32 positions of a VIMA vector to 1, while the rest of the vector is full of zeros, as depicted in Figure 4.12. If the instances' size is equal to or greater than the VIMA vector, this transformation will not be necessary. The algorithm isolates each instance inside a VIMA vector by iterating through features that one instance presents, as illustrated in Algorithm 4.5.

To facilitate the instances' vector multiply by the mask operation, we added padding to isolate the lasts instances, as shown in line 3 of the Algorithm 4.5, thus avoiding compilation errors and out-of-bounds accesses. Isolating one instance per VIMA vector enables executing all the operations mentioned above (subtraction, multiplication, and accumulated sum) more simply with better data reuse inside VIMA cache.

Algorithm 4.5: Iteration on a VIMA vector isolating the instances with a mask.

```
1  #define VSIZE 2048
2
3  __v32f *instances = (__v32f*)malloc((base_size+VSIZE)*sizeof(__v32f));
4  __v32f *mask = (__v32f*)malloc(VSIZE*sizeof(__v32f));
5  __v32f *result = (__v32f*)malloc(VSIZE*sizeof(__v32f));
6
7  for (int i = 0; i < n_features; ++i) {
8      mask[i] = 1.0;
9  }
10
11 for (int i = 0; i < base_size; i += n_features) {
12     _vim2K_fmuls(&instances[i], mask, result);
13 }
```

All the accumulated sums between each test instance and the set of training instances (calculated with VIMA routines) are stored in a different vector in memory. Afterward, the x86 square root instruction must be applied to this vector, resulting in the Euclidean Distances. This vector will store all the distances calculated between all the tests and training instances. Thus, this vector will store several Euclidean Distances equal to the number of training instances for each test instance.

Finally, in the classification step, all the distances for each test instance are paired with the label vector to find the k lowest distances. In this phase, the main interest is in the labels of the k lowest values. The label with the majority among these k lowest values is the label assigned to the test instance. This final step does not use Intrinsics-VIMA functions.

4.3.3 Multilayer Perceptron (MLP) Migration

First of all, as mentioned before, we implemented a naive NN. Our hidden layer is half of the input layer size due to its responsibility in defining relations between relevant features. It must have a balanced amount of neurons compared to the number of analyzed features in an instance. Suppose the hidden layer presents too few or too many neurons. In that case, it may not correctly identify the relevant features or consider every feature as relevant, resulting in accuracy loss during classification. The output layer has only two neurons to classify instances as either positive or negative, as depicted in Figure 4.6. As mentioned before, here we consider only the inference part of this algorithm, with pre-trained weights.

As in kNN, just a single instance is operated simultaneously, despite of more than one instance can be stored in a VIMA vector. Thus, they must be isolated with a mask to be computed. If the instances' size is equal to or greater than the VIMA vector, this transformation will not be necessary.

Each instance feature is a neuron of the input layer, which must be multiplied by the weights connections between input and hidden layers to obtain the hidden layer's activation values. As each input neuron has different connections with the neurons in the hidden layer, as depicted in Figure 4.6, this operation can be seen as a vector-matrix multiplication, where the input layer is a vector, and the weights configure a matrix as depicted in Figure 4.13. Each line of the matrix is a set of weights that must be multiplied by the input vector. As the number of neurons in the hidden layer is half of the input layer, this number represents the multiple sets of weights.

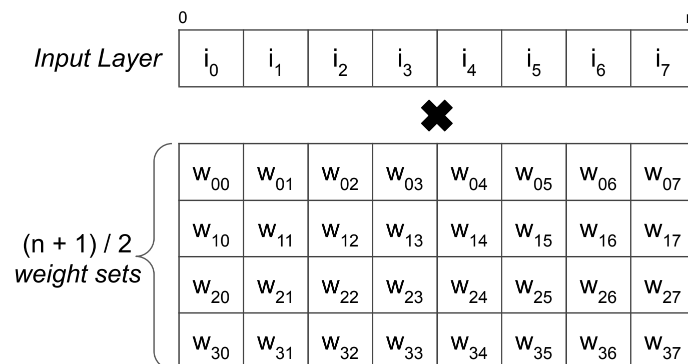


Figure 4.13: Example of a multiplication between input layer and sets of weights. In this case, the instances present 8 features.

Fortunately, we can vectorize most of these operations with Intrinsics-VIMA. Thus, the algorithm sequentially loads sets of weights into VIMA vectors. We isolate each set with a mask

before being operated with the instances. After isolating both instances and set of weights, the following routines are executed:

- *_vim2K_fmuls()* to multiply the input features and weight values (w_{xy});
- *_vim2K_fcums()* to sum all results to find out the activation value of a neuron.

This algorithm repeats until it calculates the activation value of each neuron in the hidden layer and is illustrated in Figure 4.14. The same logic of Algorithm 4.5 is applied to isolate each instance and each set of weights in VIMA vectors.

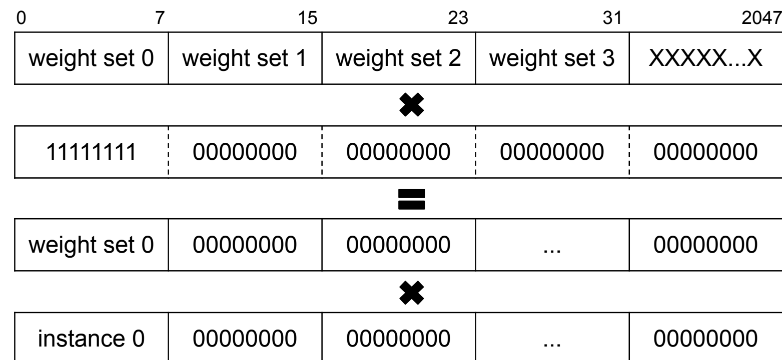


Figure 4.14: Example of a VIMA vector with four sets of weights for instances representing 8 features.

These results are all stored sequentially, for each instance, in a vector corresponding to the hidden layer activation values. After these operations finish for every instance, the hidden layer vector is added to a bias vector to adjust the activation values and reduce classification errors. Then the activation function is applied to it. In this case, the Rectified Linear Units (ReLU) function was used to apply non-linearity into the activation values. Thus zeros replace negative values.

As illustrated in Algorithm 4.6, both described operations can be implemented with the Intrinsics-VIMA library, with the following routines:

- *_vim2K_fmovs()* to initialize the bias and ReLU vectors, the first one initialized with ones and the second with zeros;
- *_vim2K_fadds()* to add the values of hidden layer and bias vectors; and
- *_vim2K_fmaxs()* to apply ReLU activation function to the values of hidden layer vector.

Algorithm 4.6: Adding a bias vector to the hidden layer activation values.

```
1  #define VSIZE 2048
2
3  __v32f *bias = (__v32f*)malloc(VSIZE * sizeof(__v32f));
4  __v32f *relu = (__v32f*)malloc(VSIZE * sizeof(__v32f));
5
6  _vim2K_fmovs(1.0, bias);
7  _vim2K_fmovs(0.0, relu);
8
9  for (int i = 0; i < hlayer_size; i += VSIZE) {
10     _vim2K_fadds(&hidden_layer[i], bias, &hidden_layer[i]);
11 }
12
13 for (int i = 0; i < hlayer_size; i += VSIZE) {
14     _vim2K_fmaxs(&hidden_layer[i], relu, &hidden_layer[i]);
15 }
```

Afterward, the activation values' calculation in the output layer is similar to the hidden layer values computation. The difference is related to the number of neurons and weights. In this case, the hidden layer size is half of the input layer, and the output layer is composed of 2 neurons only (we have only two labels). Thus only two sets of weights (w'_{xy}) are defined, both sets with the same size as the hidden layer instance.

In this manner, the algorithm requires a mask with half of the size to isolate hidden layer instances and the sets of weights to execute the operations mentioned before to calculate the output layer's two activation values. Finally, a Softmax activation function (Bishop, 2006) must be applied to them to transform these values into probabilities. The higher probability corresponds to the label most likely to classify the instance. This final step does not use Intrinsics-VIMA functions.

All the algorithms mentioned in this chapter are present in the appendix at the end of this document. Both versions, VIMA and AVX 512, are shown in Appendixes G, H, C, I, J, D, E, K, and F.

5 EXPERIMENTAL EVALUATION OF VIMA

In this chapter, we present the methodology and the simulation results for the ML kernel implementations.

5.1 METHODOLOGY AND SIMULATION SETUP

To simulate and evaluate this proposal, we adopted OrCS. Inside OrCS we modeled VIMA, a custom smart-memory architecture with FUs, a cache memory, and configurable operation size as mentioned in Chapter 2. This modeling can help researchers understand architectural behavior when executing the selected benchmarks using multiple generated statistics. Table 5.1 shows the main parameters used for this model.

Table 5.1: Baseline and VIMA system configuration.

OoO Execution Cores 32 cores @ 2.0 GHz, 32 nm; Power: 6W/core; 6-wide issue; Buffers: 18-entry fetch, 28-entry decode; 168-entry ROB; MOB entries: 64-read, 36-write; 2-load, 1-store units (1-1 cycle); 3-alu, 1-mul. and 1-div. int. units (1-3-32 cycle); 1-alu, 1-mul. and 1-div. fp. units (3-5-10 cycle); 1 branch per fetch; Branch predictor: Two-level GAs. 4096 entry BTB;
L1 Data + Inst. Cache 64 KB, 8-way, 2-cycle; 64 B line; LRU policy; Dynamic energy: 194pJ per line access; Static power: 30mW;
L2 Cache 256 KB, 8-way, 10-cycle; 64 B line; LRU policy; Dynamic energy: 340pJ per line access; Static power: 130mW;
LLC Cache 16 MB, 16-way, 22-cycle; 64 B line; LRU policy; Dynamic energy: 3.01nJ per line access; Static power: 7W;
3D Stacked Mem. 32 vaults, 8 DRAM banks/vault, 256 B row buffer; Closed-row policy; 4 GB; DRAM@1666 MHz; 4-links@8 GHz; 8 B burst width at 2.5:1 core-to-bus freq. ratio; DRAM: CAS, RP, RCD, RAS and CWD latency (9-9-9-24-7 cycles); Avg. energy per access: x86:10.8pJ/bit; VIMA:4.8pJ/bit; Static power 4W;
VIMA Processing Logic Operation frequency: 1 GHz; Power: 3.2W; 32 nm; 256 int. units: alu, mul. and div. (8-12-28 cycles); 256 fp. units: alu, mul. and div. (13-13-28 cycles); VIMA cache: 64 KB, 2-cycle (1-tag, 1-per data); Static power: 134mW; For 256 B vectors: 32 lines, 4-way; For 8 KB vectors: 8 lines, fully assoc; Dynamic energy: 1.67nJ per 256 B vector access; 53.7nJ per 8 KB vector access;

x86 baseline: The baseline architecture was inspired by the Intel Sandy Bridge processor micro-architecture and referred to as "x86". The ISA was modeled with AVX-512 instruction set capabilities besides all x86 ISA instructions. Furthermore, a 3D-stacked memory was used as the main memory.
VIMA architectures: To provide two scenarios for comparison, the proposal uses near-data operations over vectors of 256 B and 8 KB. In this approach, we implemented the NEON ISA near-data. The x86 processor triggers these VIMA instructions.
Benchmark: In our experiments, we evaluate kNN, MLP, and convolution kernels. In general, the test varies in parameters as detailed below:

- **Convolution:** We tested the algorithm with VIMA vectors of 256 B and 8 KB, a square matrix with different sizes: 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, 128 MB, 156 MB, and 512 MB. The equivalent dimensions are 512×512, 724×724, 1024×1024, 1448×1448, 2048×2048, 2896×2896, 4096×4096, 5792×5792, 8192×8192, and 11648×11648.

- **kNN:** We tested the algorithm with VIMA vectors of 256 B and 8 KB, with 4096, 8192, 16384, 32768, and 65536 training instances; 256 test instances, with 8, 16, 32, 64, 128, 256, 512, 1024, 2048 and 4096 features; and 9 neighbors.

- **MLP:** We tested the algorithm with VIMA vectors of 256 B and 8 KB, with 4096 instances and with 8, 16, 32, 64, 128, 256, 512, 1024, 2048, and 4096 features.

The evaluation focuses on architecture efficiency, not on the accuracy of each classification algorithm. Thus, the results are showed in terms of speedup and energy savings.

5.2 BEST CONDITIONS TO ACHIEVE HIGH PERFORMANCE

Before showing the speedup and energy results, it is essential to understand the best conditions in which each algorithm is expected to achieve a better performance than the baseline. This clarity helps to interpret the results. NDP architectures are best suited for specific application domains that present data streaming behavior (avoiding massive data movement) and a low data reuse ratio (the opposite takes advantage of big processor caches). Therefore, VIMA improvements are enhanced when the amount of data in memory exceeds the cache memory size. As the system configuration shown in Table 5.1, the Last-Level Cache (LLC) data cache is 16 MB in size. Full x86 applications start to suffer when their data does not fit in the cache hierarchy, causing more cache lines replacements and, consequently, a longer execution time and higher energy consumption.

Thus, in this section, we present some algorithm characteristics to explain the final results. Three different scenarios are presented, where kNN is the best case for VIMA, and the convolution is the worst.

5.2.1 k-Nearest Neighbors

As shown in Table 5.2, the kNN memory footprint with different parameters of features and instances and, in light gray, it shows in which cases we expect VIMA to present better performance (where data exceeds cache memory size). As the number of instances and features increase, the sooner it extrapolates cache memory size. As the algorithm remains the same for VIMA 256 B, 8 KB, and AVX512 implementations, the memory footprint is similar in the three cases, and the speedup and energy savings also follows this pattern, as presented below.

kNN is an algorithm with a quadratic complexity since each test instance must be calculated against the whole training dataset. This operation pattern can easily replace all the x86 cache memory during the execution, depending on the number of features and instances. Besides, massive training datasets can be larger than cache memory size. For example, the case with 4096 instances with 4096 features each one already has 16 MB in size considering just the storage of the training dataset, and it causes low reuse of data, making the baseline worse than VIMA in these cases.

Table 5.2: kNN Memory Footprint approximation for VIMA 256B, 8KB and AVX512.

	kNN - Memory Footprint (MB)				
#feat / #inst	**4096**	**8192**	**16384**	**32768**	**65536**
8	0.1	0.3	0.6	1.1	2.3
16	0.3	0.5	1.1	2.1	4.3
32	0.5	1.0	2.1	4.1	8.3
64	1.0	2.0	4.1	8.1	16.3
128	2.0	4.0	8.1	16.1	32.3
256	4.0	8.0	16.1	32.1	64.3
512	8.0	16.0	32.1	64.1	128.3
1024	16.0	32.0	64.1	128.1	256.3
2048	32.0	64.0	128.1	256.1	512.3
4096	64.0	128.0	256.1	512.1	1024.3

5.2.2 Multilayer Perceptron

MLP memory footprint presented in Table 5.3 shows a pattern similar to kNN, indicating in light gray the cases where data exceeds cache memory size. As the algorithm remains the same for VIMA 256 B, 8 KB, and AVX512 implementations, the memory footprint is also similar in the three cases. However, this pattern does not occur for speedup and energy savings.

MLP, as we implemented, has a linear complexity considering the use of the instances, since the vector with instances is multiplied by a defined number of weights. The amount of memory needed to allocate the sets of weights increases as the number of features increases and is not affected by the number of instances. For example, regardless of the number of instances, considering the case with 4096 features, just the sets of weights will occupy 8 MB in size. Thus, low data reuse will only happen when increasing the number of features while considering auxiliary arrays and structures allocated during execution.

Table 5.3: MLP Memory Footprint approximation for VIMA 256B, 8KB and AVX512.

	MLP - Memory Footprint (MB)				
#feat / #inst	**4096**	**8192**	**16384**	**32768**	**65536**
8	0.1	0.1	0.3	0.5	1.0
16	0.1	0.3	0.5	1.0	2.0
32	0.3	0.5	1.0	2.0	4.0
64	0.5	1.0	2.0	4.0	8.0
128	1.0	2.0	4.0	8.0	16.0
256	2.1	4.1	8.1	16.1	32.1
512	4.5	8.5	16.5	32.5	64.5
1024	10.0	18.0	34.0	66.0	130.0
2048	24.0	40.0	72.0	136.0	264.0
4096	64.0	96.0	160.0	288.0	544.0

5.2.3 Convolution

Unlike the previous cases, convolution is present in terms of size (in MB) to compute. For example, 1 MB is equivalent to a square matrix with 512 lines and 512 columns considering

4 bytes of data representation. However, even though we describe the square matrix as being 1 MB in size, the algorithm consumes double the size during execution since two matrices are allocated (the actual and the new matrix) to execute the operation. Table 5.4 present the convolution's memory footprint, where the cells in light gray indicate the matrices sizes that exceed the cache memory size.

By itself, we expect that matrices with sizes greater than 16 MB still present medium data reuse in the cache memory, as at most three matrix lines present data reuse at a time.

Table 5.4: Convolution Memory Footprint approximation for VIMA 256B, 8KB and AVX512.

Matrix Size (MB)	**Convolution Memory Footprint (MB)**	**Matrix line size (KB)**
1	2.0	2.0
2	4.0	2.8
4	8.0	4.0
8	16.0	5.6
16	32.0	8.0
32	64.0	11.3
64	128.0	16.0
128	256.0	22.6
256	512.0	32.0
512	1024.0	44.2

Considering the pattern in Figure 4.8 in Chapter 4, it is possible to see that 3 different lines are required to compute the convolution of those cells. Considering these 9 accesses, initially at most 3 cache misses will occur, but 6 cache hits will follow these. Among the 3 lines, 2 will be reused to compute the cells below, resulting in 2 more hits and only one miss. Thus, a poor data reuse is expected only when a matrix line is at least 16 MB, so it would not be possible to reuse data from other lines to calculate the convolution. In this case, the greater matrix size tested was 512 MB with a dimension of 11648×11648, leading to lines of 44.2 KB in size. Therefore, it is challenging to achieve speedups and energy gain in convolution using VIMA, as the x86 cache memory will fast delivery data during reuse.

5.3 EXECUTION TIME RESULTS

In this section, we will start to show the results for execution time. Figures 5.1~5.5 present speedup results for the kNN algorithm varying the number of instances. When there is a VIMA slowdown compared to x86, the values are negative. When there is a VIMA speedup compared to x86, the values are positive. Thus, the first thing to observe is the slowdown of every configuration for the implementation with 256 B VIMA vectors, presented in white bars. In this configuration, VIMA's slows down execution by up to 9× compared to x86. As the memory footprint increases, the slowdowns oscillate, and when it exceeds cache memory size, it stabilizes, reaching a maximum of 3× slower than x86. This low performance occurs due to the 3D-stacked memory bandwidth, as explained in Chapter 2, since the instructions over 256 B vectors will not use the whole bandwidth.

kNN starts to present better results for VIMA with 8 KB vectors when increasing memory usage, as shown in Figure 5.2. Note that both variables, number of features, and instances affect the final memory footprint. Thus, it is possible to observe that the results improve as the number of instances increases. In this configuration, VIMA's speeds up execution by up to 11× compared to x86.

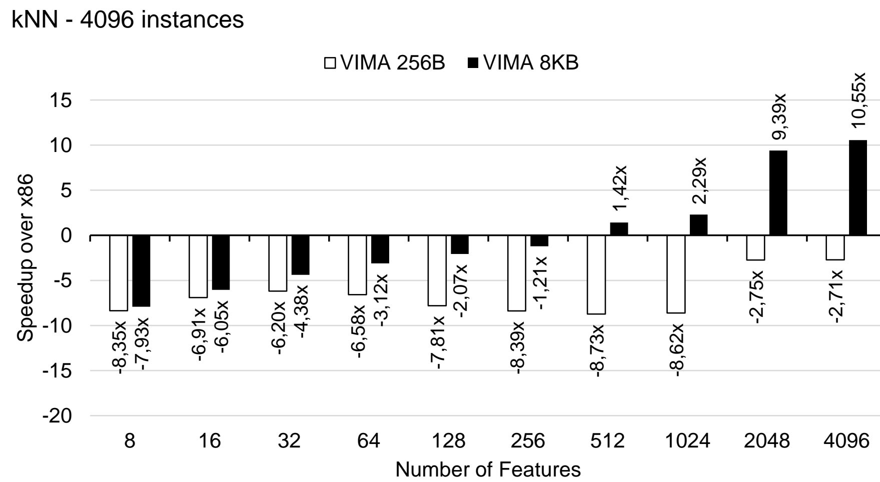


Figure 5.1: VIMA's speedup over x86. Comparing implementations using both sizes of VIMA vectors when executing with 4096 instances and varying the number of features.

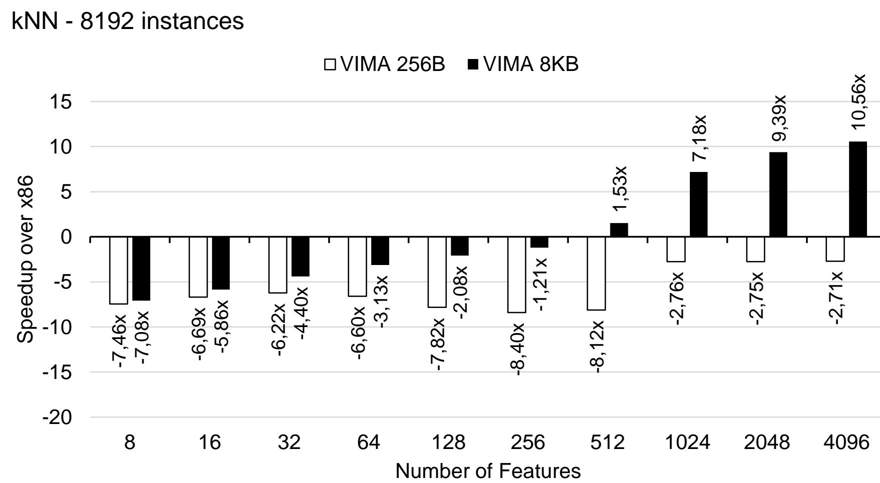


Figure 5.2: VIMA's speedup over x86. Comparing implementations using both sizes of VIMA vectors when executing with 8192 instances and varying the number of features.

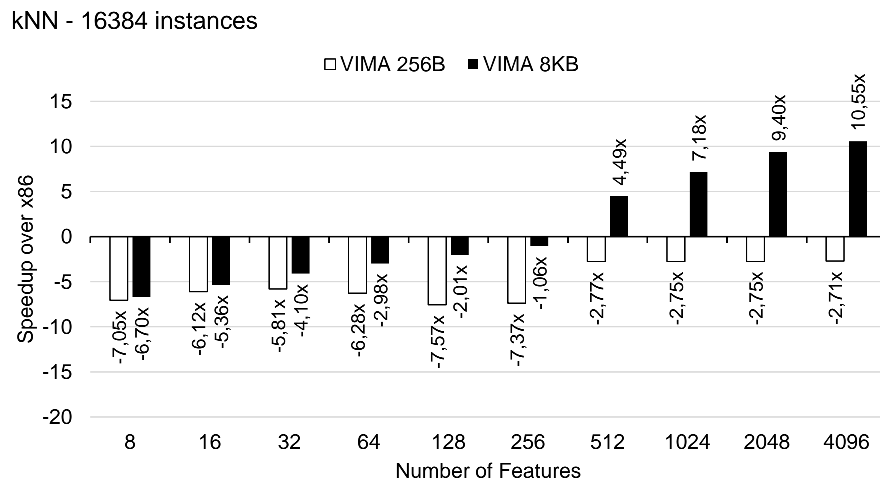


Figure 5.3: VIMA's speedup over x86. Comparing implementations using both sizes of VIMA vectors when executing with 16384 instances and varying the number of features.

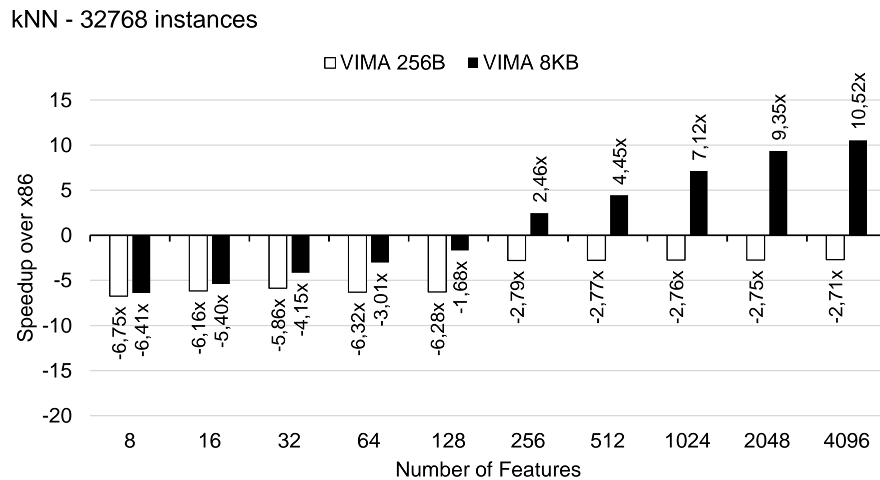


Figure 5.4: VIMA's speedup over x86. Comparing implementations using both sizes of VIMA vectors when executing with 32768 instances and varying the number of features.

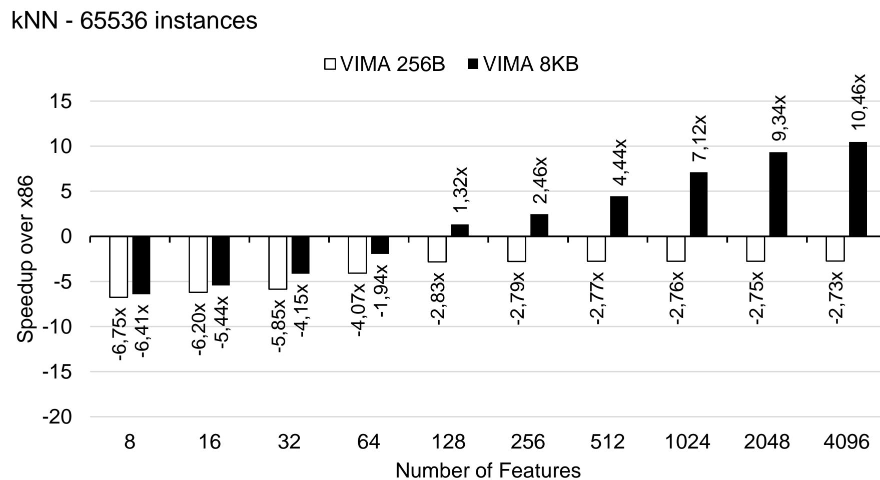


Figure 5.5: VIMA's speedup over x86. Comparing implementations using both sizes of VIMA vectors when executing with 65536 instances and varying the number of features.

MLP speedup is shown in Figure 5.6. As explained before, only the number of features influence the results for MLP. The number of instances does not influence because each instance is independent and used a single time only. Thus, we present results just for MLP with 4096 instances.

As kNN, the implementation with VIMA vectors with 256 B in size is inefficient, reaching up to 15× of slowdown compared to baseline. The implementation with 8 KB VIMA vector starts to show a slight speedup for instances with 512 features and has a significant increase with 4096 features, reaching up to 11× of speedup compared to baseline.

We can observe that until 128 features, the slowdown for both implementations (256 B and 8 KB) is similar. In these cases, just one VIMA vector is used to store the instances. With 128 features, we require two 256 B VIMA vectors to store an instance, which explains the slight decrease in performance. Similarly, the slowdown for 256 B VIMA vectors worsens from 256 to 2048 features since many more 256 B VIMA vectors are needed to store the instances and to operate with a vast amount of weights.

The baseline still makes fair use of x86 cache memory, and that is the reason for the moderated speedups with VIMA vectors of 8 KB. With 4096 features, the baseline starts to

present worse performance due to low data reuse in the cache memory, making the speedup for 8 KB VIMA vectors significantly better, also reducing the slowdown for VIMA vectors of 256 B in size.

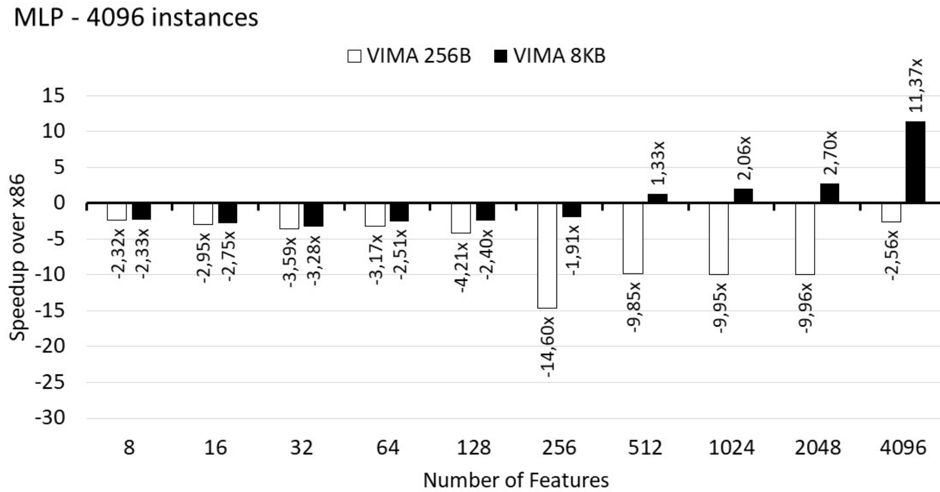


Figure 5.6: VIMA's speedup over x86. Comparing implementations using both sizes of VIMA vectors when executing with 4096 instances and varying the number of features.

Figure 5.7 presents speedup results for the convolution algorithm described on Algorithm 4.4 with matrices from size 512×512 to 11648×11648. As we explained the algorithms above, the convolution implementation using VIMA 256 B only presents slowdowns, reaching a maximum of 23× slowdown over baseline. As the slowdowns, the speedups for VIMA 8 KB implementation are not linear since it depends on the usage of vector and the execution time of the x86 baseline. We expected better results considering a larger matrix of 2048x2048 that occupies a total of 16 MB of memory, but the application still makes fair usage of the cache hierarchy of x86. Nevertheless, the maximum performance achieved was a speedup of up to 3× for a matrix with 32 MB in size.

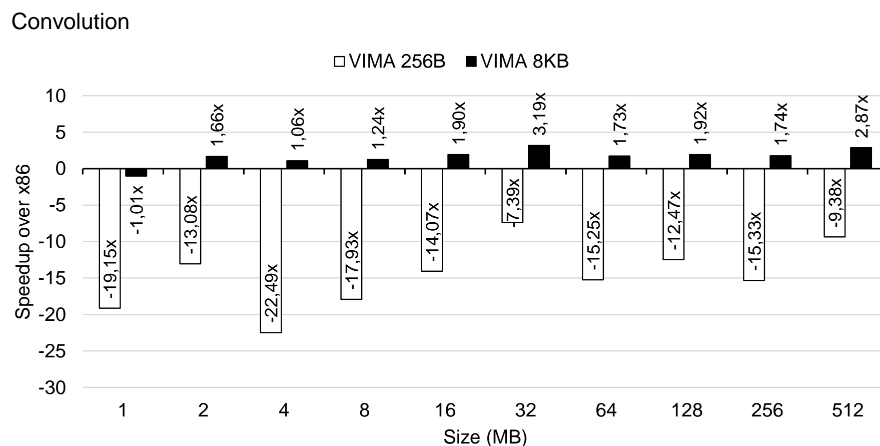


Figure 5.7: VIMA's speedup over x86. Comparing implementations using both sizes of VIMA vectors when varying matrix size.

5.4 ENERGY RESULTS

Figures 5.8~5.12 present the energy efficiency for kNN, which is proportional to the speedup. Again, we can observe that the implementation with 256 B VIMA vectors is more expensive than x86 implementation, spending up to 12× more than x86. As the speedup graphics, the energy gains oscillates with smaller memory footprint sizes and stabilizes when it exceeds the cache memory size, spending around 3× more than x86.

However, it is possible to reduce energy consumption by up to 8× using 8 KB VIMA vectors compared to the baseline. For tiny amounts of data, this vector size spends more than 256 B VIMA vectors, reaching a cost of up to 13× higher than baseline. Thus, there are no energy savings for kNN with a small number of features and instances.

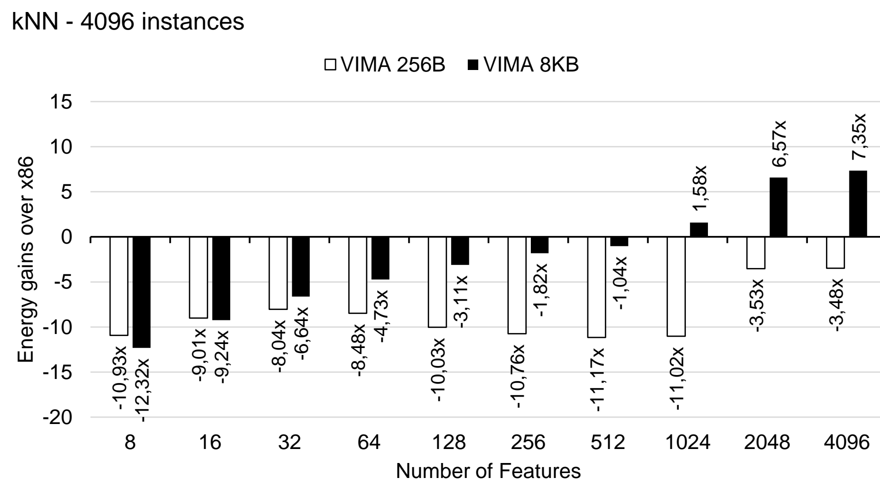


Figure 5.8: VIMA's energy consumption over x86. Comparing implementations using both sizes of VIMA vectors when executing with 4096 instances and varying the number of features.

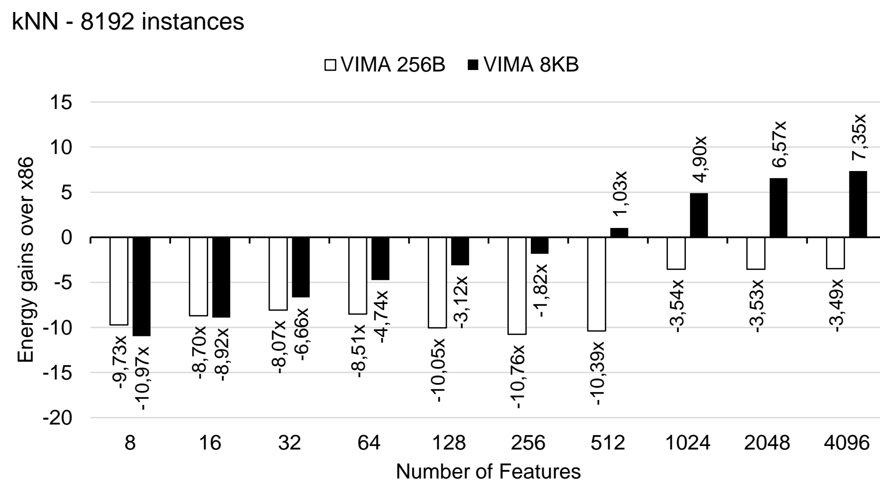


Figure 5.9: VIMA's energy consumption over x86. Comparing implementations using both sizes of VIMA vectors when executing with 8192 instances and varying the number of features.

Figure 5.6 shows the energy consumption of the MLP application executed with 4096 instances. Although energy consumption follows the speedup pattern, it is possible to notice that the energy spent in slowdown cases is slightly higher than the values achieved in Figure 5.6. In

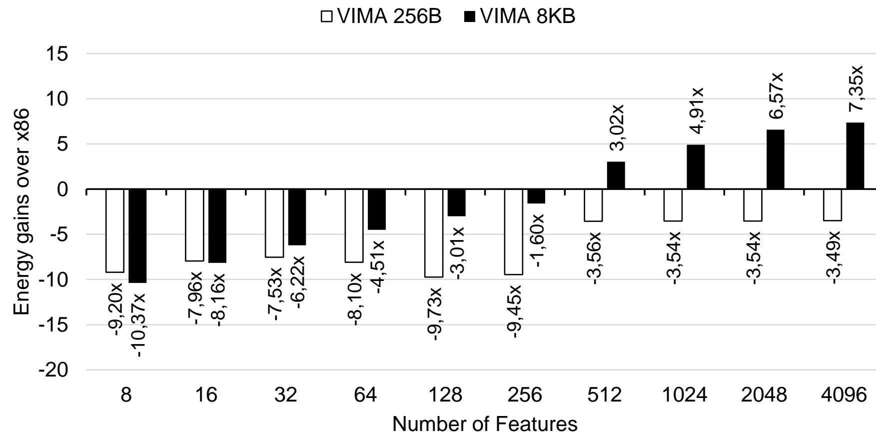


Figure 5.10: VIMA's energy consumption over x86. Comparing implementations using both sizes of VIMA vectors when executing with 16384 instances and varying the number of features.

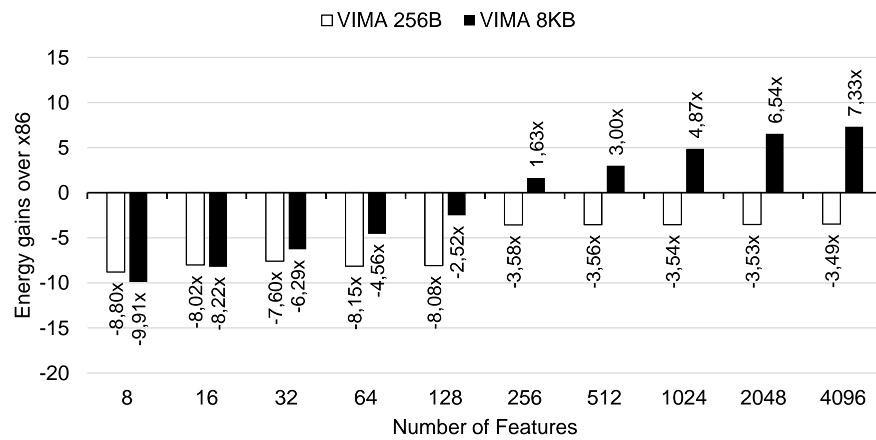


Figure 5.11: VIMA's energy consumption over x86. Comparing implementations using both sizes of VIMA vectors when executing with 32768 instances and varying the number of features.

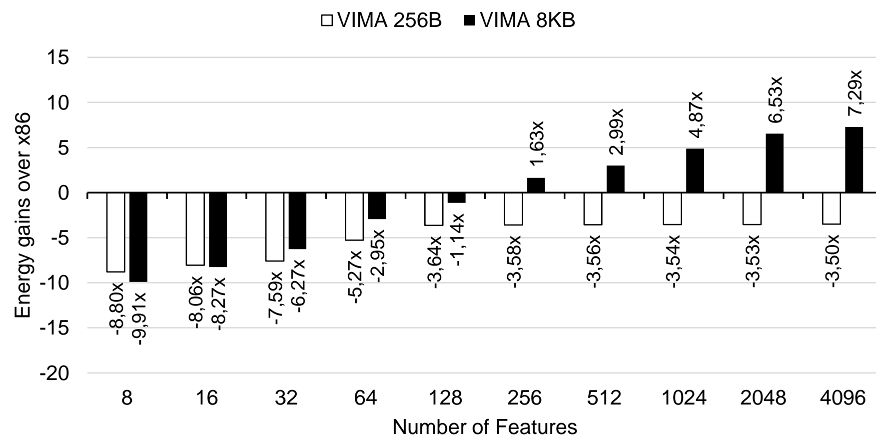


Figure 5.12: VIMA's energy consumption over x86. Comparing implementations using both sizes of VIMA vectors when executing with 65536 instances and varying the number of features.

contrast, the energy spent in good speedup cases is slightly lower. Moreover, this difference is more significant in the last case, for instances with 4096 features.

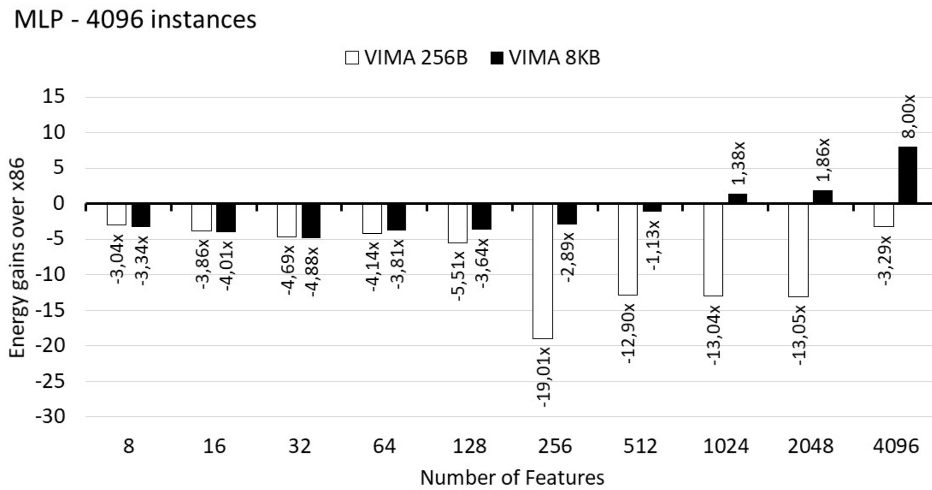


Figure 5.13: VIMA's energy consumption over x86. Comparing implementations using both sizes of VIMA vectors when executing with 4096 instances and varying the number of features.

Figure 5.14 shows the energy consumption for the convolution algorithm, which follows the speedup pattern considering that energy is always slightly lower than the speedup. For the implementation with VIMA 256 B, the energy consumption is always higher than the baseline, and these values reduce in the first test with a memory footprint larger than cache memory size. The energy consumption for VIMA 8 KB implementation presents an oscillation, mainly in lower speedup cases, and the gains are higher when the matrix size exceeds the cache memory size, spending just half of the energy compared to the baseline.

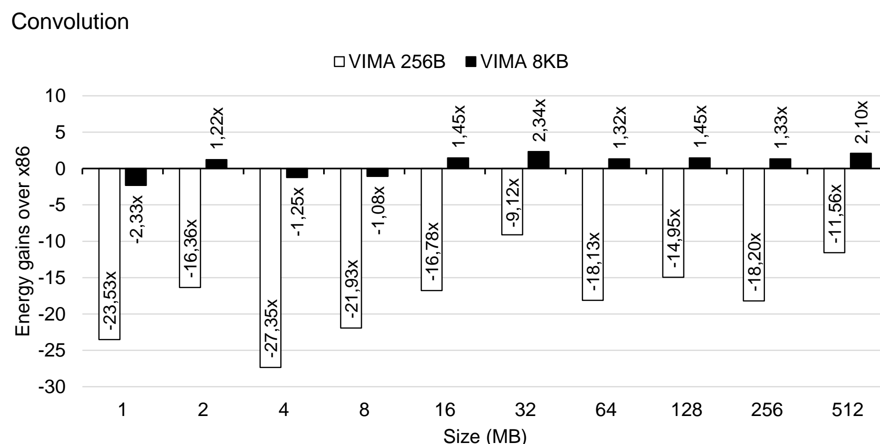


Figure 5.14: VIMA's energy consumption over x86. Comparing implementations using both sizes of VIMA vectors varying the matrix size.

The energy savings achieved by VIMA depend directly on memory usage following the speedup trend. Whenever the memory footprint fits inside the x86 cache memory, the processor presents higher efficiency. Otherwise, VIMA consumes less due to faster execution and less data movement. This result reinforces the concept that NDP must be seen as an accelerator for applications with data-stream behavior and low data reuse.

6 FINAL CONSIDERATIONS

After analyzing our proposal results, we would like to point out some concluding remarks about our findings and also related to possible improvements as future work.

First of all, we decided to evaluate ML applications since its high demand nowadays and because of the enormous amount of data processed by these algorithms. However, our methodology implied the need to write the code in C/C++ language, which is not a common task for data scientists, which usually relies on already established libraries and routines in Python language. Thus, we understand that this work focus is not on ML or Data Science, which already has a series of concerns in handling data and algorithms, so they do not have to worry about implementing mathematical methods and NNs from scratch. Instead, our work focuses more on the architectural interface for high-performance computing and ML in this context.

Besides, during the development of our work, it became clear the fact that it is not only ML algorithms that can benefit from VIMA. Nevertheless, VIMA may improve a series of algorithms that rely on similar characteristics, such as algorithms that deal with massive datasets exceeding conventional cache memory hierarchy size and perform none or short-term data reuse. In the beginning, we wanted to implement a complete version of the algorithms to deal with different dataset sizes. Also, implementing a CNN with training and inference stages. However, we realize that this may not be possible due to time and resource restraints. We rely on generating simulation traces, and this step can be time-consuming depending on the algorithm's complexity. As VIMA was being thought together with this dissertation, we decided to keep the algorithms as simple as possible. In this manner, they may not entirely resemble a ML algorithm, so we emphasize that any algorithm with similar complexity and structure can benefit from VIMA.

Regarding the architectural details, our methodology is straightforward. It can be easily applied to evaluate any algorithm in VIMA or any architecture supported by OrCS and any ISA supported by the trace generator used here. Besides, it is possible to extend both tools with different and even non-existent architectures and ISAs. As a curiosity, our team remodeled Intrinsics-VIMA a few times. At first, we have implemented the functions to operate over 32- and 64-bits integer representation only. Then, as we defined this dissertation's scope, we implemented the same function, but to operate over 64-bits floating-point representation. After understanding better ML algorithms, we decided to implement the same functions to operate over 32-bits floating-point representation. Finally, when most of VIMA was implemented, we decide to check out which instructions were being represented in these Intrinsics functions in order to follow ARM NEON ISA. As future work, we consider extending Intrinsics-VIMA to operate over 16-bits floating-point and 8-bits integer representation since newer NN are using these representations to reduce code complexity and also it can achieve better performance in VIMA.

When implementing codes with Intrinsics libraries, it is required low effort from the developer, who can still compile, execute, and debug the code as usual. To implement codes with Intrinsics-VIMA is necessary to adapt the code to fully use the VIMA vectors to achieve better performance in VIMA. In our case, when we are working with shorter instances, we read in a vector as many instances as possible and isolate them with a mask to implement most of the code with Intrinsics-VIMA. However, another alternative is to repeat an instance until it fills the vector to operate with the remaining data. It was an initial approach found in the firsts commits of the codes in GitHub[1]. Nevertheless, it proved that it is necessary to consider the aspects of VIMA

[1]https://github.com/ascordeiro/intrinsics

itself since its cache memory size is small, so we have to implement the code focusing on VIMA vectors reuse to achieve better performance.

For the results, as we mentioned, we expected to achieve better performance with VIMA when x86 performs poorly due to high cache miss ratios (due to big datasets). During the evaluations, we understand that the pattern of memory access of the algorithm also can influence VIMA's performance in comparison with x86. For instance, the convolution algorithm makes fair data reuse in x86 cache memory. In contrast, the kNN algorithm data reuse in x86 cache worsens as soon as the number of instances and features increases. Also, in the beginning, the idea was to compare VIMA with GPU since it is vastly used for ML algorithms. However, to do so, it was necessary to make a fair comparison. In this case, the idea was to implement a GPU in OrCS and compare the architectural results, as we did for x86 with the AVX-512 instructions. Due to time constraints, it was not possible, so we decided to make an indirect comparison with GPU. In this case, we searched for related work that compared the same algorithms used in this dissertation, in an x86 architecture and a GPU. We selected more recent work that relies on x86 architectures, which are more similar to Sandy Bridge. Thus, we selected the following works:

- Skryjomski et al. (2019) made a comparison of the kNN algorithm between a CPU Intel Xeon E5-2690v4 with 2.60GHz, and a GPU NVIDIA GeForce 1080Ti with @1885MHz. The authors also varied the number of features and instances. Their results report speedup up to 200× with the GPU over the CPU;

- A Dawwd and M AL Layla (2015) made a comparison of the MLP algorithm between a CPU Intel Core i7-2670QMCPU with 2.20GHz, and a GPU GeForce 610M. They also varied the dataset's size and obtained a speedup of up to 300× with the GPU over the CPU.

- Siklosi et al. (2018) made a comparison of the stencil algorithm between a CPU Intel Xeon E5-2660v4 with 2.00GHz and GPUs NVIDIA P100 and NVIDIA V100, and they obtained a speedup of up to 20× with the GPU over the CPU.

Considering these related work, we can observe that VIMA is 30× slower than a GPU. Nevertheless, it consumes 60× less energy. We can understand how VIMA compares against a GPU using this indirect and shallow comparison. Besides, it is good to reinforce that we are not trying to compete with other architectures. We are just establishing parameters to understand better VIMA's performance. A CPU has a competitive performance when it can make fair reuse of data inside the cache hierarchy. At the same time, a GPU can achieve high performance for a series of algorithms, and it is evident that it also consumes a significant amount of energy to achieve this high performance. Nevertheless, we consider VIMA very competitive in terms of energy consumption.

Finally, it is necessary to highlight that we are giving an estimate only in this dissertation due to the naive implementation of the algorithms and the simulation of a non-existent architecture in the real world, which is in its firsts versions. Thus, this research is essential to understand the VIMA's viability to define the next steps better. As a parameter, in the real world, the kNN algorithm can be used in satellite imagery and online games to calculate the distance between players. In these cases, it is common to use fewer features and a significant amount of data. The MLP or CNN algorithms can be applied in face recognition and recommendation systems, which are vastly applied in social network and applications such as YouTube, Netflix, and Spotify. They work with a considerable amount of data, and each person is an instance.

7 CONCLUSION

Considering the memory-wall and dark silicon problems, several approaches to NDP are emerging in the last years. Concurrently, ML algorithms are getting more relevant when analyzing ever-increasing volumes of data. In this dissertation, we linked both problems to propose the migration of ML kernels to a vector execution near-data system to achieve a high execution speedup with low energy consumption.

The ML kernels were simulated in OrCS, using the Instrinsics-VIMA library as a tool to easy the code portability. We achieved a speedup of up to 10× for kNN, 11× for MLP, and 3× for convolution. Meanwhile, we obtained energy savings of 7× for kNN, 8× for MLP, and 3× for convolution compared to a baseline x86 system.

The results rely on the algorithm behavior, memory footprint, and computational performance of VIMA and x86 architectures. For example, memory footprint sizes greater than LLC memory can drop the baseline performance as it executes more cache line replacements (due to higher cache misses), spending more energy and execution time in comparison to VIMA. However, different implementation of the algorithm or its memory access pattern can change this scenario by better using cache memory, resulting in a competitive computational performance for the baseline.

As future work, we consider extending the migration to other ML algorithms, including its training phase. We also consider evaluating multi-threaded applications using VIMA operations.

All the source code for our VIMA architecture simulation, the ML algorithms, and the Intrinsics-VIMA library are freely available in our on-line repositories[1][2].

[1]https://github.com/mazalves/
[2]https://github.com/ascordeiro/

REFERENCES

A Dawwd, S. and M AL Layla, N. (2015). Training acceleration of multi-layer perceptron using multicore cpu and gpu under matlab environment. *AL-Rafdain Engineering Journal (AREJ)*, 23(3):136–148.

Afonso, S., Acosta, A., and Almeida, F. (2017). Automatic acceleration of stencil codes in android devices. In *Int. Conf. on Algorithms and Architectures for Parallel Processing*.

Agrawal, A., Ankit, A., and Roy, K. (2018). Spare: Spiking neural network acceleration using rom-embedded rams as in-memory-computation primitives. *IEEE Transactions on Computers*.

Ahn, J., Hong, S., Yoo, S., Mutlu, O., and Choi, K. (2016). A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Computer Architecture News*, 43(3):105–117.

Alpaydin, E. (2009). *Introduction to machine learning*. MIT press.

Alves, M. A. Z. (2014). *Increasing Energy Efficiency of Processor Caches via Line Usage Predictors*. PhD thesis, Universidade Federal do Rio Grande do Sul.

Alves, M. A. Z., Diener, M., Santos, P. C., and Carro, L. (2016). Large vector extensions inside the hmc. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1249–1254. IEEE.

Alves, M. A. Z., Villavieja, C., Diener, M., Moreira, F. B., and Navaux, P. O. A. (2015). Sinuca: A validated micro-architecture simulator. In *HPCC/CSS/ICESS*, pages 605–610.

AMD (2015). DDR5 and HBM comparison. `https://www.amd.com/system/files/documents/high-bandwidth-memory-hbm.pdf`. [Online; accessed 01-July-2019].

Angizi, S., He, Z., Rakin, A. S., and Fan, D. (2018). Cmp-pim: an energy-efficient comparator-based processing-in-memory neural network accelerator. In *Proceedings of the 55th Annual Design Automation Conference*, page 105. ACM.

Azarkhish, E., Rossi, D., Loi, I., and Benini, L. (2018). Neurostream: Scalable and energy efficient deep learning with smart memory cubes. *IEEE Transactions on Parallel & Distributed Systems*, pages 1–1.

Bartlett, M. S., Littlewort, G., Lainscsek, C., Fasel, I., and Movellan, J. (2004). Machine learning methods for fully automatic recognition of facial expressions and facial actions. In *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No. 04CH37583)*, volume 1, pages 592–597. IEEE.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

Bojnordi, M. N. and Ipek, E. (2016). Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–13. IEEE.

Boroumand, A., Ghose, S., Kim, Y., Ausavarungnirun, R., Shiu, E., Thakur, R., Kim, D., Kuusela, A., Knies, A., Ranganathan, P., et al. (2018). Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Cadambi, S., Majumdar, A., Becchi, M., Chakradhar, S., and Graf, H. P. (2010). A programmable parallel accelerator for learning and classification. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 273–284. ACM.

Chen, F., Song, L., and Chen, Y. (2018). Regan: A pipelined reram-based accelerator for generative adversarial networks. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 178–183. IEEE.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794.

Cheng, M., Xia, L., Zhu, Z., Cai, Y., Xie, Y., Wang, Y., and Yang, H. (2017). Time: A training-in-memory architecture for memristor-based deep neural networks. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 26. ACM.

Cheng, M., Xia, L., Zhu, Z., Cai, Y., Xie, Y., Wang, Y., and Yang, H. (2018). Time: A training-in-memory architecture for rram-based deep neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):834–847.

Chi, P., Li, S., Xu, C., Zhang, T., Zhao, J., Liu, Y., Wang, Y., and Xie, Y. (2016). Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ACM SIGARCH Computer Architecture News*, pages 27–39. IEEE Press.

Coorporation, I. (2009). Intel 64 and ia-32 architectures optimization reference manual.

Cordeiro, A. S., Kepe, T. R., Tomé, D. G., de Almeida, E. C., and Alves, M. A. Z. (2017). Intrinsics-hmc: An automatic trace generator for simulations of processing-in-memory instructions. *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*.

de Lima, J. P. C., Santos, P. C., de Moura, R. F., Alves, M. A., Beck, A. C., and Carro, L. (2019). Exploiting reconfigurable vector processing for energy-efficient computation in 3d-stacked memories. In *International Symposium on Applied Reconfigurable Computing*, pages 262–276. Springer.

Deng, Q., Jiang, L., Zhang, Y., Zhang, M., and Yang, J. (2018). Dracc: a dram based accelerator for accurate cnn inference. In *Proceedings of the 55th Annual Design Automation Conference*, page 168. ACM.

Deng, Q., Zhang, Y., Zhang, M., and Yang, J. (2019). Lacc: Exploiting lookup table-based fast and accurate vector multiplication in dram-based cnn accelerator. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 128. ACM.

Dietterich, T. G. (2000). Ensemble methods in machine learning. In *Int. workshop on multiple classifier systems*.

Efnusheva, D., Cholakoska, A., and Tentov, A. (2017). A survey of different approaches for overcoming the processor-memory bottleneck. *International Journal of Computer Science and Information Technology*, 9(2):151–163.

Elliott, D. G., Stumm, M., Snelgrove, W. M., Cojocaru, C., and McKenzie, R. (1999a). Computational ram: Implementing processors in memory. *IEEE Design & Test of Computers*, 16(1):32–41.

Elliott, D. G., Stumm, M., Snelgrove, W. M., et al. (1999b). Computational RAM: Implementing Processors in Memory. *Design and Test of Computers*, 16(1):32–41.

Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., and Burger, D. (2011). Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*, pages 365–376. IEEE.

Fan, D. and Angizi, S. (2017). Energy efficient in-memory binary deep neural network accelerator with dual-mode sot-mram. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 609–612. IEEE.

Ganguly, A., Singh, V., Muralidhar, R., and Fujita, M. (2018). Memory-system requirements for convolutional neural networks. In *Proceedings of the International Symposium on Memory Systems*, pages 291–197. ACM.

Gantz, J. and Reinsel, D. (2011). Extracting value from chaos. *IDC iview*, 1142(2011):1–12.

Gantz, J. and Reinsel, D. (2012). The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007(2012):1–16.

Gao, D., Shen, T., and Zhuo, C. (2018). A design framework for processing-in-memory accelerator. In *2018 ACM/IEEE International Workshop on System Level Interconnect Prediction (SLIP)*, pages 1–6. IEEE.

Gao, M., Ayers, G., and Kozyrakis, C. (2015). Practical near-data processing for in-memory analytics frameworks. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 113–124. IEEE.

Gao, M., Pu, J., Yang, X., Horowitz, M., and Kozyrakis, C. (2017). Tetris: Scalable and efficient neural network acceleration with 3d memory. *ACM SIGOPS Operating Systems Review*, 51(2):751–764.

Gardner, M. W. and Dorling, S. (1998). Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric environment*, 32(14-15).

Géron, A. (2019). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media.

Gupta, S., Imani, M., Kaur, H., and Rosing, T. S. (2019). Nnpim: A processing in-memory architecture for neural network acceleration. *IEEE Transactions on Computers*.

Gupta, S., Imani, M., and Rosing, T. (2018). Felix: Fast and energy-efficient logic in memory. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7. IEEE.

Hashemi, M., Ebrahimi, E., Mutlu, O., Patt, Y. N., et al. (2016). Accelerating dependent cache misses with an enhanced memory controller. In *Int. Symp. on Computer Architecture (ISCA)*.

Holewinski, J., Pouchet, L.-N., and Sadayappan, P. (2012). High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320.

Hong, B., Ro, Y., and Kim, J. (2018). Multi-dimensional parallel training of winograd layer on memory-centric architecture. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 682–695. IEEE.

Hrusca, J. (2015). PIM comparison. `https://www.extremetech.com/computing/197720-beyond-ddr4-understand-the-differences-between-wide-io-hbm-and-hybrid-memory-cube`. [Online; accessed 01-July-2019].

Hybrid Memory Cube Consortium (2013). Hybrid memory cube specification rev. 2.0. http://www.hybridmemorycube.org/.

Hybrid Memory Cube Consortium (2014). Hybrid memory cube specification 2.1. http://www.hybridmemorycube.org/.

Imani, M., Gupta, S., Kim, Y., and Rosing, T. (2019a). Floatpim: In-memory acceleration of deep neural network training with high precision. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 802–815. ACM.

Imani, M., Gupta, S., Kim, Y., Zhou, M., and Rosing, T. (2019b). Digitalpim: Digital-based processing in-memory for big data acceleration. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pages 429–434. ACM.

Imani, M., Gupta, S., and Rosing, T. (2018). Genpim: Generalized processing in-memory to accelerate data intensive applications. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1155–1158. IEEE.

Jain, R. (1990). *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons.

Jeddeloh, J. and Keeth, B. (2012). Hybrid memory cube new DRAM architecture increases density and performance. In *Symp. on VLSI Technology*.

Ji, Y., Zhang, Y., Li, S., Chi, P., Jiang, C., Qu, P., Xie, Y., and Chen, W. (2016). Neutrams: Neural network transformation and co-design under neuromorphic hardware constraints. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 21. IEEE Press.

Jiang, S., Priya, S. R., Elango, N., Clay, J., and Sridhar, R. (2019). An energy efficient in-memory computing machine learning classifier scheme. In *2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)*, pages 157–162. IEEE.

Jun, H., Nam, S., Jin, H., Lee, J.-C., Park, Y. J., and Lee, J. J. (2017). High-bandwidth memory (hbm) test challenges and solutions. *IEEE Design & Test*, 34(1):16–25.

Kaplan, R., Yavits, L., and Ginosar, R. (2018). Prins: Processing-in-storage acceleration of machine learning. *IEEE Transactions on Nanotechnology*.

Kara, K., Alistarh, D., Alonso, G., Mutlu, O., and Zhang, C. (2017). Fpga-accelerated dense linear machine learning: A precision-convergence trade-off. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 160–167. IEEE.

Kim, J. and Kim, Y. (2014). Hbm: Memory solution for bandwidth-hungry processors. In *2014 IEEE Hot Chips 26 Symposium (HCS)*, pages 1–24. IEEE.

Kourou, K., Exarchos, T. P., Exarchos, K. P., Karamouzis, M. V., and Fotiadis, D. I. (2015). Machine learning applications in cancer prognosis and prediction. *Computational and structural biotechnology journal*, 13:8–17.

Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., and Sadayappan, P. (2007). Effective automatic parallelization of stencil computations. *ACM sigplan notices*, 42(6):235–244.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*.

Kuderer, M., Gulati, S., and Burgard, W. (2015). Learning driving styles for autonomous vehicles from demonstration. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2641–2646. IEEE.

Lacassagne, L., Etiemble, D., Hassan Zahraee, A., Dominguez, A., and Vezolle, P. (2014). High level transforms for simd and low-level computer vision algorithms. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, pages 49–56.

Lawrence, S., Giles, C. L., Tsoi, A. C., and Back, A. D. (1997). Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113.

Li, B., Song, L., Chen, F., Qian, X., Chen, Y., and Li, H. H. (2018). Reram-based accelerator for deep learning. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pages 815–820. IEEE.

Li, S., Niu, D., Malladi, K. T., Zheng, H., Brennan, B., and Xie, Y. (2017). Drisa: A dram-based reconfigurable in-situ accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 288–301. ACM.

Libbrecht, M. W. and Noble, W. S. (2015). Machine learning applications in genetics and genomics. *Nature Reviews Genetics*, 16(6):321.

Lima, J. a. P., Santos, P. C., Alves, M. A. Z., Beck, A. C. S., and Carro, L. (2018). Design space exploration for pim architectures in 3d-stacked memories. In *Proceedings of the Computing Frontiers Conference*. ACM.

Liu, J., Zhao, H., Ogleari, M. A., Li, D., and Zhao, J. (2018). Processing-in-memory for energy-efficient neural network training: A heterogeneous approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 655–668. IEEE.

Lomont, C. (2011). Introduction to intel advanced vector extensions. *Intel White Paper*, pages 1–21.

Long, Y., Na, T., and Mukhopadhyay, S. (2018). Reram-based processing-in-memory architecture for recurrent neural network acceleration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–14.

Lue, H.-T., Wang, K.-C., and Lu, C.-Y. (2018). 3d and-type nvm for in-memory computing of artificial intelligence. In *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pages 1–2. IEEE.

McDanel, B., Teerapittayanon, S., and Kung, H. (2017). Embedded binarized neural networks. *arXiv preprint arXiv:1709.02260*.

Min, C., Mao, J., Li, H., and Chen, Y. (2019). Neuralhmc: an efficient hmc-based accelerator for deep neural networks. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 394–399. ACM.

Mitchell, T. M. (1997). Mcgraw-hill science. *Engineering/Math*, 1:27.

Moore, G. E. (1998). Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85.

Moore, G. E. et al. (1975). Progress in digital integrated electronics. In *Electron Devices Meeting*, volume 21, pages 11–13.

Nair, R., Antao, S. F., Bertolli, C., Bose, P., Brunheroto, J. R., Chen, T., Cher, C.-Y., Costa, C. H., Doi, J., Evangelinos, C., et al. (2015). Active memory cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59.

Nowatzyk, A., Pong, F., and Saulsbury, A. (1996). Missing the memory wall: The case for processor/memory integration. In *Int. Symp. on Computer Architecture (ISCA)*.

Nurvitadhi, E., Sim, J., Sheffield, D., Mishra, A., Krishnan, S., and Marr, D. (2016). Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE.

Oliveira, G. F., Santos, P. C., Alves, M. A., and Carro, L. (2017a). Nim: An hmc-based machine for neuron computation. In *International Symposium on Applied Reconfigurable Computing*, pages 28–35. Springer.

Oliveira, G. F., Santos, P. C., Alves, M. A. Z., and Carro, L. (2017b). A generic processing in memory cycle accurate simulator under hybrid memory cube architecture. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 54–61. IEEE.

Olmen, J. V., Mercha, A., Katti, G., et al. (2008). 3D stacked IC demonstration using a through silicon via first approach. In *Int. Electron Devices Meeting*.

Pan, Y., Ouyang, P., Zhao, Y., Kang, W., Yin, S., Zhang, Y., Zhao, W., and Wei, S. (2018a). A mlc stt-mram based computing in-memory architec-ture for binary neural network. In *2018 IEEE International Magnetics Conference (INTERMAG)*, pages 1–1. IEEE.

Pan, Y., Ouyang, P., Zhao, Y., Kang, W., Yin, S., Zhang, Y., Zhao, W., and Wei, S. (2018b). A multilevel cell stt-mram-based computing in-memory accelerator for binary convolutional neural network. *IEEE Transactions on Magnetics*, 54(11):1–5.

Patterson, D., Anderson, T., Cardwell, N., et al. (1997a). A case for intelligent RAM. *IEEE Micro*, 17(2):34–44.

Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., and Yelick, K. (1997b). A case for intelligent ram. *IEEE micro*, 17(2):34–44.

Pawlowski, J. (2011a). Hybrid memory cube (hmc). *Hot Chips*, 23.

Pawlowski, J. T. (2011b). Hybrid memory cube (hmc). In *2011 IEEE Hot chips 23 symposium (HCS)*, pages 1–24. IEEE.

Petersen, K., Vakkalanka, S., and Kuzniarz, L. (2015). Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18.

Peterson, L. E. (2009). K-nearest neighbor. *Scholarpedia*, 4(2).

Pugsley, S., Jestes, J., Balasubramonian, R., et al. (2014). Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads. *IEEE Micro*, 34(4):44–52.

Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S., et al. (2016). Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35.

Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., and Emer, J. (2007a). Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2).

Qureshi, M. K., Suleman, M. A., and Patt, Y. N. (2007b). Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *Int. Symp. on High Performance Computer Architecture (HPCA)*.

Rakotomamonjy, A. (2003). Variable selection using svm-based criteria. *Journal of machine learning research*, 3(Mar).

Raschka, S. (2015). *Python machine learning*. Packt Publishing Ltd.

Ren, D. Q. (2011). Algorithm level power efficiency optimization for cpu–gpu processing element in data intensive simd/spmd computing. *Journal of Parallel and Distributed Computing*, 71(2):245–253.

Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Salamat, S., Imani, M., Gupta, S., and Rosing, T. (2018). Rnsnet: In-memory neural network acceleration using residue number system. In *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–12. IEEE.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229.

Santos, P. C., Oliveira, G. F., Lima, J. P., Alves, M. A., Carro, L., and Beck, A. C. (2018). Processing in 3d memories to speed up operations on complex data structures. In *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*. IEEE.

Santos, P. C., Oliveira, G. F., Tomé, D. G., Alves, M. A., Almeida, E. C., and Carro, L. (2017). Operand size reconfiguration for big data processing in memory. In *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*.

Schuiki, F., Schaffner, M., Gürkaynak, F. K., and Benini, L. (2018). A scalable near-memory architecture for training deep neural networks on large in-memory datasets. *arXiv preprint arXiv:1803.04783*.

Shafiee, A., Nag, A., Muralimanohar, N., Balasubramonian, R., Strachan, J. P., Hu, M., Williams, R. S., and Srikumar, V. (2016). Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26.

Shen, J. P. and Lipasti, M. H. (2013). *Modern processor design: fundamentals of superscalar processors*. Waveland Press.

Siklosi, B., Reguly, I. Z., and Mudalige, G. R. (2018). Heterogeneous cpu-gpu execution of stencil applications. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–80. IEEE.

Sim, J., Seol, H., and Kim, L.-S. (2018). Nid: processing binary convolutional neural network in commodity dram. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE.

Skryjomski, P., Krawczyk, B., and Cano, A. (2019). Speeding up k-nearest neighbors classifier for large-scale multi-label learning on gpus. *Neurocomputing*, 354:10–19.

Smith, A. J. (1982). Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530.

Srivastava, P., Kang, M., Gonugondla, S. K., Lim, S., Choi, J., Adve, V., Kim, N. S., and Shanbhag, N. (2018). Promise: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 43–56. IEEE Press.

Sudarshan, C., Lappas, J., Ghaffar, M. M., Rybalkin, V., Weis, C., Jung, M., and Wehn, N. (2019). An in-dram neural network processing engine. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

Sukhwani, B., Min, H., Thoennes, M., Dube, P., Iyer, B., Brezzo, B., Dillenberger, D., and Asaad, S. (2012). Database analytics acceleration using fpgas. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 411–420. IEEE.

Thanh-Hoang, T., Shambayati, A., Deutschbein, C., Hoffmann, H., and Chien, A. A. (2014). Performance and energy limits of a processor-integrated fft accelerator. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE.

Thoma, Y., Dassatti, A., and Molla, D. (2013). Fpga 2: An open source framework for fpga-gpu pcie communication. In *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE.

Thottethodi, M., Vijaykumar, T., et al. (2018). Millipede: Die-stacked memory optimizations for big data machine learning analytics. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 160–171. IEEE.

Tian, Y., Pei, K., Jana, S., and Ray, B. (2018). Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314.

Toffoli, T. and Margolus, N. (1987). *Cellular automata machines: a new environment for modeling*. MIT press.

Tomé, D. G., Santos, P. C., Carro, L., Almeida, E. C., and Alves, M. A. Z. (2018). Hipe: Hmc instruction predication extension applied on database processing. In *Design, Automation & Test in Europe Conf.*

Transcend (2014). DDR comparison. `https://www.transcend-info.com/Support/FAQ-296`. [Online; accessed 01-July-2019].

Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24.

Xu, L., Zhang, D. P., and Jayasena, N. (2015). Scaling deep learning on multiple in-memory processors. In *Proceedings of the 3rd Workshop on Near-Data Processing*.

Yu, C. D., Huang, J., Austin, W., Xiao, B., and Biros, G. (2015). Performance optimization for the k-nearest neighbors kernel on x86 architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 7. ACM.

Zaitsev, D. A. (2017). A generalized neighborhood for cellular automata. *Theoretical Computer Science*, 666:21–35.

APPENDIX A – TABLE OF INTRINSICS-VIMA INSTRUCTIONS

Table A.1: Table of Intrinsics-VIMA instructions.

Begin of Table		
Function	Data type	Description
VIMA Arithmetic Instructions - Integer		
`_vim64_iadds(*a, *b, *c)`	`__v32s`	Performs signed addition between 32-bit elements source vectors A[0:63] and B[0:63] and stores the result into the destination vector C[0:63].
`_vim2K_iadds(*a, *b, *c)`	`__v32s`	Performs signed addition between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim64_iaddu(*a, *b, *c)`	`__v32u`	Performs unsigned addition between 32-bit elements source vectors A[0:63] and B[0:63] and stores the result into the destination vector C[0:63].
`_vim2K_iaddu(*a, *b, *c)`	`__v32u`	Performs unsigned addition between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim64_isubs(*a, *b, *c)`	`__v32s`	Performs signed subtraction between 32-bit elements source vectors A[0:63] and B[0:63] and stores the result into the destination vector C[0:63].
`_vim2K_isubs(*a, *b, *c)`	`__v32s`	Performs signed subtraction between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim64_isubu(*a, *b, *c)`	`__v32u`	Performs unsigned subtraction between 32-bit elements source vectors A[0:63] and B[0:63] and stores the result into the destination vector C[0:63].
`_vim2K_isubu(*a, *b, *c)`	`__v32u`	Performs unsigned subtraction between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim64_iabss(*a, *b)`	`__v32s`	Takes the absolute value of each 32-bit element in a source vector A[0:63] and stores it into the destination vector B[0:63].
`_vim2K_iabss(*a, *b)`	`__v32s`	Takes the absolute value of each 32-bit element in a source vector A[0:2047] and stores it into the destination vector B[0:2047].
`_vim64_imaxs(*a, *b, *c)`	`__v32s`	Find the maximal value between each 32-bit element of source vectors A[0:63] and B[0:63] and stores it into the destination vector C[0:63].
`_vim2K_imaxs(*a, *b, *c)`	`__v32s`	Find the maximal value between each 32-bit element of source vectors A[0:2047] and B[0:2047] and stores it into the destination vector C[0:2047].
`_vim64_imins(*a, *b, *c)`	`__v32s`	Find the minimal value between each 32-bit element of source vectors A[0:63] and B[0:63] and stores it into the destination vector C[0:63].
`_vim2K_imins(*a, *b, *c)`	`__v32s`	Find the minimal value between each 32-bit element of source vectors A[0:2047] and B[0:2047] and stores it into the destination vector C[0:2047].
`_vim64_icpys(*a, *b)`	`__v32s`	Performs signed copy of 32-bit element of source vectors A[0:63] and stores it into the destination vector B[0:63].
`_vim2K_icpys(*a, *b)`	`__v32s`	Performs signed copy of 32-bit element of source vectors A[0:2047] and stores it into the destination vector B[0:2047].
`_vim64_icpyu(*a, *b)`	`__v32u`	Performs unsigned copy of 32-bit element of source vectors A[0:63] and stores it into the destination vector B[0:63].

Continuation of Table A.1		
Function	Data type	Description
`_vim2K_icpyu(*a, *b)`	`__v32u`	Performs unsigned copy of 32-bit element of source vectors A[0:2047] and stores it into the destination vector B[0:2047].
VIMA Arithmetic Instructions - Floating-point Single Precision		
`_vim64_fadds(*a, *b, *c)`	`__v32f`	Performs signed addition between 32-bit elements source vectors A[0:63] and B[0:63] and stores the result into the destination vector C[0:63].
`_vim2K_fadds(*a, *b, *c)`	`__v32f`	Performs signed addition between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim64_fsubs(*a, *b, *c)`	`__v32f`	Performs signed subtraction between 32-bit elements source vectors A[0:63] and B[0:63] and stores the result into the destination vector C[0:63].
`_vim2K_fsubs(*a, *b, *c)`	`__v32f`	Performs signed subtraction between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim64_fabss(*a, *b)`	`__v32f`	Takes the absolute value of each 32-bit element in a source vector A[0:63] and stores it into the destination vector B[0:63].
`_vim2K_fabss(*a, *b)`	`__v32f`	Takes the absolute value of each 32-bit element in a source vector A[0:2047] and stores it into the destination vector B[0:2047].
`_vim64_fmaxs(*a, *b, *c)`	`__v32f`	Find the maximal value between each 32-bit element of source vectors A[0:63] and B[0:63] and stores it into the destination vector C[0:63].
`_vim2K_fmaxs(*a, *b, *c)`	`__v32f`	Find the maximal value between each 32-bit element of source vectors A[0:2047] and B[0:2047] and stores it into the destination vector C[0:2047].
`_vim64_fmins(*a, *b, *c)`	`__v32f`	Find the minimal value between each 32-bit element of source vectors A[0:63] and B[0:63] and stores it into the destination vector C[0:63].
`_vim2K_fmins(*a, *b, *c)`	`__v32f`	Find the minimal value between each 32-bit element of source vectors A[0:2047] and B[0:2047] and stores it into the destination vector C[0:2047].
`_vim64_fcpys(*a, *b)`	`__v32f`	Performs signed copy of 32-bit element of source vectors A[0:63] and stores it into the destination vector B[0:63].
`_vim2K_fcpys(*a, *b)`	`__v32f`	Performs signed copy of 32-bit element of source vectors A[0:2047] and stores it into the destination vector B[0:2047].
VIMA Arithmetic Instructions - Floating-point Double Precision		
`_vim32_dadds(*a, *b, *c)`	`__v64d`	Performs signed addition between 64-bit elements source vectors A[0:31] and B[0:31] and stores the result into the destination vector C[0:31].
`_vim1K_dadds(*a, *b, *c)`	`__v64d`	Performs signed addition between 64-bit elements source vectors A[0:1023] and B[0:1023] and stores the result into the destination vector C[0:1023].
`_vim32_dsubs(*a, *b, *c)`	`__v64d`	Performs signed subtraction between 64-bit elements source vectors A[0:31] and B[0:31] and stores the result into the destination vector C[0:31].
`_vim1K_dsubs(*a, *b, *c)`	`__v64d`	Performs signed subtraction between 64-bit elements source vectors A[0:1023] and B[0:1023] and stores the result into the destination vector C[0:1023].

Continuation of Table A.1

Function	Data type	Description
_vim32_dabss(*a, *b)	__v64d	Takes the absolute value of each 64-bit element in a source vector A[0:31] and stores it into the destination vector B[0:31].
_vim1K_dabss(*a, *b)	__v64d	Takes the absolute value of each 64-bit element in a source vector A[0:1023] and stores it into the destination vector B[0:1023].
_vim32_dmaxs(*a, *b, *c)	__v64d	Find the maximal value between each 64-bit element of source vectors A[0:31] and B[0:31] and stores it into the destination vector C[0:31].
_vim1K_dmaxs(*a, *b, *c)	__v64d	Find the maximal value between each 64-bit element of source vectors A[0:1023] and B[0:1023] and stores it into the destination vector C[0:1023].
_vim32_dmins(*a, *b, *c)	__v64d	Find the minimal value between each 64-bit element of source vectors A[0:31] and B[0:31] and stores it into the destination vector C[0:31].
_vim1K_dmins(*a, *b, *c)	__v64d	Find the minimal value between each 64-bit element of source vectors A[0:1023] and B[0:1023] and stores it into the destination vector C[0:1023].
_vim32_dcpys(*a, *b)	__v64d	Performs signed copy of 64-bit element of source vectors A[0:31] and stores it into the destination vector B[0:31].
_vim1K_dcpys(*a, *b)	__v64d	Performs signed copy of 64-bit element of source vectors A[0:1023] and stores it into the destination vector B[0:1023].
VIMA Logic Instructions - Integer		
_vima2K_iandu(*a, *b, *c)	__vm32u	Performs AND operation between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
_vim2K_iandu(*a, *b, *c)	__v32u	Performs AND operation between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
_vim64_iorun(*a, *b, *c)	__v32u	Performs OR operation between 32-bit elements source vectors A[0:63] and B[0:63] and stores the result into the destination vector C[0:63].
_vim2K_iorun(*a, *b, *c)	__v32u	Performs OR operation between 32-bit elements of source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
_vim64_ixoru(*a, *b, *c)	__v32u	Performs XOR operation between 32-bit elements source vectors A[0:63] and B[0:63] and stores the result into the destination vector C[0:63].
_vim2K_ixoru(*a, *b, *c)	__v32u	Performs XOR operation between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
_vim64_inots(*a, *b)	__v32s	Performs NOT operation in 32-bit elements source vector A[0:63] and stores the result into the destination vector B[0:63].
_vim2K_inots(*a, *b)	__v32s	Performs NOT operation in 32-bit elements source vector A[0:2047] and stores the result into the destination vector B[0:2047].
VIMA Comparison Instructions - Integer		
_vim64_islts(*a, *b, *c)	__v32s	Compare each signed 32-bit elements from source vectors A[0:63] and B[0:63] and if the element of A[0:63] is minor, then destination source C[0:63] stores 1 in the same position, otherwise, stores 0.

Continuation of Table A.1

Function	Data type	Description
`_vim2K_islts(*a, *b, *c)`	`__v32s`	Compare each signed 32-bit elements from source vectors A[0:2047] and B[0:2047] and if the element of A[0:2047] is minor, then destination source C[0:2047] stores 1 in the same position, otherwise, stores 0.
`_vim64_isltu(*a, *b, *c)`	`__v32u`	Compare each unsigned 32-bit elements from source vectors A[0:63] and B[0:63] and if the element of A[0:63] is minor, then destination source C[0:63] stores 1 in the same position, otherwise, stores 0.
`_vim2K_isltu(*a, *b, *c)`	`__v32u`	Compare each unsigned 32-bit elements from source vectors A[0:2047] and B[0:2047] and if the element of A[0:2047] is minor, then destination source C[0:2047] stores 1 in the same position, otherwise, stores 0.
`_vim64_icmqs(*a, *b, *c)`	`__v32s`	Compare each signed 32-bit elements from source vectors A[0:63] and B[0:63] and if they are equal, then destination source C[0:63] stores 1 in the same position, otherwise, stores 0.
`_vim2K_icmqs(*a, *b, *c)`	`__v32s`	Compare each signed 32-bit elements from source vectors A[0:2047] and B[0:2047] and if they are equal, then destination source C[0:2047] stores 1 in the same position, otherwise, stores 0.
`_vim64_icmqu(*a, *b, *c)`	`__v32u`	Compare each unsigned 32-bit elements from source vectors A[0:63] and B[0:63] and if they are equal, then destination source C[0:63] stores 1 in the same position, otherwise, stores 0.
`_vim2K_icmqu(*a, *b, *c)`	`__v32u`	Compares each unsigned 32-bit elements from source vectors A[0:2047] and B[0:2047] and if they are equal, then destination source C[0:2047] stores 1 in the same position, otherwise, stores 0.
VIMA Comparison Instructions - Floating-point Single Precision		
`_vim64_fslts(*a, *b, *c)`	`__v32f`	Compare each signed 32-bit elements from source vectors A[0:63] and B[0:63] and if the element of A[0:63] is minor, then destination source C[0:63] stores 1 in the same position, otherwise, stores 0.
`_vim2K_fslts(*a, *b, *c)`	`__v32f`	Compare each signed 32-bit elements from source vectors A[0:2047] and B[0:2047] and if the element of A[0:2047] is minor, then destination source C[0:2047] stores 1 in the same position, otherwise, stores 0.
`_vim64_fcmqs(*a, *b, *c)`	`__v32f`	Compare each signed 32-bit elements from source vectors A[0:63] and B[0:63] and if they are equal, then destination source C[0:63] stores 1 in the same position, otherwise, stores 0.
`_vim2K_fcmqs(*a, *b, *c)`	`__v32f`	Compare each signed 32-bit elements from source vectors A[0:2047] and B[0:2047] and if they are equal, then destination source C[0:2047] stores 1 in the same position, otherwise, stores 0.
VIMA Comparison Instructions - Floating-point Double Precision		
`_vim32_dslts(*a, *b, *c)`	`__v64d`	Compare each signed 64-bit elements from source vectors A[0:31] and B[0:31] and if the element of A[0:31] is minor, then destination source C[0:31] stores 1 in the same position, otherwise, stores 0.
`_vim1K_dslts(*a, *b, *c)`	`__v64d`	Compare each signed 64-bit elements from source vectors A[0:1023] and B[0:1023] and if the element of A[0:1023] is minor, then destination source C[0:1023] stores 1 in the same position, otherwise, stores 0.
`_vim32_dcmqs(*a, *b, *c)`	`__v64d`	Compare each signed 64-bit elements from source vectors A[0:31] and B[0:31] and if they are equal, then destination source C[0:31] stores 1 in the same position, otherwise, stores 0.
`_vim1K_dcmqs(*a, *b, *c)`	`__v64d`	Compare each signed 64-bit elements from source vectors A[0:1023] and B[0:1023] and if they are equal, then destination source C[0:1023] stores 1 in the same position, otherwise, stores 0.
VIMA Shift Instructions - Integer		
`_vim64_isllu(*a, *b, *c)`	`__v32u`	Left shift each 32-bit element in source vector A[0:63] the amount specified in source vector B[0:63] and stores the result into the destination vector C[0:63]. This operation does not shift signal.

Continuation of Table A.1

Function	Data type	Description
`_vim2K_isllu(*a, *b, *c)`	`__v32u`	Left shift each 32-bit element in source vector A[0:2047] the amount specified in source vector B[0:2047] and stores the result into the destination vector C[0:2047]. This operation does not shift signal.
`_vim64_isrlu(*a, *b, *c)`	`__v32u`	Right shift each 32-bit element in source vector A[0:63] the amount specified in source vector B[0:63] and stores the result into the destination vector C[0:63]. This operation does not shift signal.
`_vim2K_isrlu(*a, *b, *c)`	`__v32u`	Right shift each 32-bit element in source vector A[0:2047] the amount specified in source vector B[0:2047] and stores the result into the destination vector C[0:2047]. This operation does not shift signal.
`_vim64_isras(*a, *b, *c)`	`__v32s`	Right shift each 32-bit element in source vector A[0:63] the amount specified in source vector B[0:63] and stores the result into the destination vector C[0:63]. This operation shifts signal.
`_vim2K_isras(*a, *b, *c)`	`__v32s`	Right shift each 32-bit element in source vector A[0:2047] the amount specified in source vector B[0:2047] and stores the result into the destination vector C[0:2047]. This operation shifts signal.
VIMA Multiplication/Division Instructions - Integer		
`_vim64_idivs(*a, *b, *c)`	`__v32s`	Performs a signed division between 32-bit elements from source vectors A[0:63] and B[0:63] and stores the result into the destination vector C[0:63].
`_vim2K_idivs(*a, *b, *c)`	`__v32s`	Performs a signed division between 32-bit elements from source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim64_idivu(*a, *b, *c)`	`__v32u`	Performs an unsigned division between 32-bit elements from source vectors A[0:63] and B[0:63] and stores the result into the destination vector C[0:63].
`_vim2K_idivu(*a, *b, *c)`	`__v32u`	Performs an unsigned division between 32-bit elements from source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim64_imuls(*a, *b, *c)`	`__v32s`	Performs signed multiplication between 32-bit elements from source vectors A[0:63] and B[0:63] and stores the result into the destination vector C[0:63].
`_vim2K_imuls(*a, *b, *c)`	`__v32s`	Performs signed multiplication between 32-bit elements from source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim64_imulu(*a, *b, *c)`	`__v32u`	Performs unsigned multiplication between 32-bit elements from source vectors A[0:63] and B[0:63] and stores the result into the destination vector C[0:63].
`_vim2K_imulu(*a, *b, *c)`	`__v32u`	Performs unsigned multiplication between 32-bit elements from source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim32_imuls(*a, *b, *c)`	`__v64s`	Performs signed multiplication between 64-bit elements from source vectors A[0:31] and B[0:31] and stores the result into the destination vector C[0:31].
`_vim1K_imuls(*a, *b, *c)`	`__v64s`	Performs signed multiplication between 64-bit elements from source vectors A[0:1023] and B[0:1023] and stores the result into the destination vector C[0:1023].
`_vim32_imulu(*a, *b, *c)`	`__v64u`	Performss unsigned multiplication between 64-bit elements from source vectors A[0:31] and B[0:31] and stores the result into the destination vector C[0:31].
`_vim1K_imulu(*a, *b, *c)`	`__v64u`	Performs unsigned multiplication between 64-bit elements from source vectors A[0:1023] and B[0:1023] and stores the result into the destination vector C[0:1023].

Continuation of Table A.1		
Function	Data type	Description
`_vim64_icums(*a, *b)`	`__v32s`	Performs signed accumulated sum between 32-bit elements from source vector A[0:63] and stores the result into the destination variable b.
`_vim2K_icums(*a, *b)`	`__v32s`	Performs signed accumulated sum between 32-bit elements from source vector A[0:2047] and stores the result into the destination variable b.
`_vim64_icumu(*a, *b)`	`__v32u`	Performss unsigned accumulated sum between 32-bit elements from source vector A[0:63] and stores the result into the destination variable b.
`_vim2K_icumu(*a, *b)`	`__v32u`	Performs unsigned accumulated sum between 32-bit elements from source vector A[0:2047] and stores the result into the destination variable b.
VIMA Multiplication/Division Instructions - Floating-point Single Precision		
`_vim64_fdivs(*a, *b, *c)`	`__v32f`	Performs a signed division between 32-bit elements from source vectors A[0:63] and B[0:63] and stores the result into the destination vector C[0:63].
`_vim2K_fdivs(*a, *b, *c)`	`__v32f`	Performs a signed division between 32-bit elements from source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim64_fmuls(*a, *b, *c)`	`__v32f`	Performs signed multiplication between 32-bit elements from source vectors A[0:63] and B[0:63] and stores the result into the destination vector C[0:63].
`_vim2K_fmuls(*a, *b, *c)`	`__v32f`	Performs signed multiplication between 32-bit elements from source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim64_fcums(*a, *b)`	`__v32f`	Performs signed accumulated sum between 32-bit elements from source vector A[0:63] and stores the result into the destination variable b.
`_vim2K_fcums(*a, *b)`	`__v32f`	Performs signed accumulated sum between 32-bit elements from source vector A[0:2047] and stores the result into the destination variable b.
VIMA Multiplication/Division Instructions - Floating-point Double Precision		
`_vim32_ddivs(*a, *b, *c)`	`__v64d`	Performs a signed division between 64-bit elements from source vectors A[0:31] and B[0:31] and stores the result into the destination vector C[0:31].
`_vim1K_ddivs(*a, *b, *c)`	`__v64d`	Performs a signed division between 64-bit elements from source vectors A[0:1023] and B[0:1023] and stores the result into the destination vector C[0:1023].
`_vim32_dmuls(*a, *b, *c)`	`__v64d`	Performs signed multiplication between 64-bit elements from source vectors A[0:31] and B[0:31] and stores the result into the destination vector C[0:31].
`_vim1K_dmuls(*a, *b, *c)`	`__v64d`	Performs signed multiplication between 64-bit elements from source vectors A[0:1023] and B[0:1023] and stores the result into the destination vector C[0:1023].
`_vim32_dcums(*a, *b)`	`__v64d`	Performs signed accumulated sum between 64-bit elements from source vector A[0:31] and stores the result into the destination variable b.
`_vim1K_dcums(*a, *b)`	`__v64d`	Performs signed accumulated sum between 64-bit elements from source vector A[0:1023] and stores the result into the destination variable b.
VIMA Immediate Instructions - Integer		

Continuation of Table A.1		
Function	Data type	Description
_vim64_imovs(*a, *b)	__v32s	Replicate a signed 32-bit immediate b into the vector A[0:63].
_vim2K_imovs(*a, *b)	__v32s	Replicate a signed 32-bit immediate b into the vector A[0:2047].
_vim64_imovu(*a, *b)	__v32u	Replicate a unsigned 32-bit immediate b into the vector A[0:63].
_vim2K_imovu(*a, *b)	__v32u	Replicate a unsigned 32-bit immediate b into the vector A[0:2047].
VIMA Immediate Instructions - Floating-point Single Precision		
_vim64_fmovs(*a, *b)	__v32f	Replicate a signed 32-bit immediate b into the vector A[0:63].
_vim2K_fmovs(*a, *b)	__v32f	Replicate a signed 32-bit immediate b into the vector A[0:2047].
VIMA Immediate Instructions - Floating-point Double Precision		
_vim32_dmovs(*a, *b)	__v64d	Replicate a signed 64-bit immediate b into the vector A[0:31].
_vim1K_dmovs(*a, *b)	__v64d	Replicate a signed 64-bit immediate b into the vector A[0:1023].
End of Table		

APPENDIX B – TABLE OF MAPPING STUDY

Table B.1: Table of selected papers.

Begin of Table						
ID	Paper title	Author names	Year	Institution	Country	Search Engine
1	Exploiting Parallelism for Convolutional Connections in Processing-In-Memory Architecture	Yi Wang, Mingxu Zhang, Jing Yang	2017	Shenzhen University, Harbin Institute of Technology	China	IEEE Xplore, ACM
2	Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach	Jiawen Liu, Hengyu Zhao, Matheus A. Ogleari, Dong Li, Jishen Zhao	2018	University of California	EUA	IEEE Xplore
3	A scalable processing-in-memory accelerator for parallel graph processing	Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, Kiyoung Choi	2015	Seoul National University, Oracle Labs, Carnegie Mellon University	Korea, EUA	IEEE Xplore, ACM
4	A Design Framework for Processing-In-Memory Accelerator	Di Gao, Tianhao Shen, Cheng Zhuo	2018	Zhejiang University	China	IEEE Xplore, ACM
5	Similarity Search on Automata Processors	Vincent T. Lee, Justin Kotalik, Carlo C. del Mundo, Armin Alaghi, Luis Ceze, Mark Oskin	2017	University of Washington	EUA	IEEE Xplore
6	Multi-dimensional Parallel Training of Winograd Layer on Memory-Centric Architecture	Byungchul Hong, Yeonju Ro, John Kim	2018	KAIST School of Electrical Engineering	Korea	IEEE Xplore
7	A programmable parallel accelerator for learning and classification	Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, Hans Peter Graf	2010	NEC Laboratories America	EUA	IEEE Xplore, ACM
8	Fine-Grained Task Migration for Graph Algorithms Using Processing in Memory	Paula Aguilera, Dong Ping Zhang, Nam Sung Kim, Nuwan Jayasena	2016	University of Wisconsin-Madison, AMD Research, University of Illinois	EUA	IEEE Xplore
9	Millipede: Die-Stacked Memory Optimizations for Big Data Machine Learning Analytics	Nitin, Mithuna Thottethodi, T. N. Vijaykumar	2018	NVIDIA, Purdue University	EUA	IEEE Xplore
10	In-Memory Distributed Matrix Computation Processing and Optimization	Yongyang Yu, Mingjie Tang, Walid G. Aref, Qutaibah M. Malluhi, Mostafa M. Abbas, Mourad Ouzzani	2017	Purdue University, Qatar University, Qatar Computing Research Institute	Qatar, EUA	IEEE Xplore
11	Optimization of Real-World MapReduce Applications With Flame-MR: Practical Use Cases	Jorge Veiga, Roberto R. Expósito, Bruno Raffin, Juan Touriño	2018	Universidade da Coruña, Université Grenoble Alpes	Espanha, França	IEEE Xplore

Continuation of Table B.1

ID	Paper title	Author names	Year	Institution	Country	Search Engine
12	DRISA: a DRAM-based Reconfigurable In-Situ Accelerator	Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, Yuan Xie	2017	University of California, Samsung Semiconductor Inc.	EUA	ACM
13	Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks	Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, Onur Mutlu	2018	Carnegie Mellon University, Seoul National University, Google, Samsung Research, ETH Zürich	EUA, Korea, Switzer-land	ACM
14	NID: processing binary convolutional neural network in commodity DRAM	Jaehyeong Sim, Hoseok Seol, Lee-Sup Kim	2018	Korea Advanced Institute of Science and Technology,	Korea	ACM
15	DrAcc: a DRAM based accelerator for accurate CNN inference	Quan Deng, Lei Jiang, Youtao Zhang, Minxuan Zhang, Jun Yang	2018	National University of Defense Technology, Indiana University Bloomington, University of Pittsburgh	EUA	ACM
16	Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes	Erfan Azarkhish, Davide Rossi, Igor Loi, Luca Benini	2017	University of Bologna, Swiss Federal Institute of Technology Zurich	Italy, Switzer-land	IEEE Xplore
17	A Scalable Near-Memory Architecture for Training Deep Neural Networks on Large In-Memory Datasets	Fabian Schuiki, Michael Schaffner, Frank Gurkaynak, Luca Benini	2018	D-ITET - ETH Zurich, Microelectronics Design Center - ETH Zurich, Universita degli Studi di Bologna Scuola di Ingegneria e Architettura	Italy, Switzer-land	IEEE Xplore
18	Processor-in-memory support for artificial neural networks	Joshua Schabel, Lee Baker, Sumon Dey, Weifu Li, Paul D. Franzon	2016	North Carolina State University	EUA	IEEE Xplore
19	NeuralHMC: an efficient HMC-based accelerator for deep neural networks	Chuhan Min, Jiachen Mao, Hai Li, Yiran Chen	2019	University of Pittsburgh, Duke University	EUA	ACM
20	Memory-system requirements for convolutional neural networks	Kevin Siu, Dylan Malone Stuart, Mostafa Mahmoud, Andreas Moshovos	2018	University of Toronto	EUA	IEEE Xplore, ACM
21	Scaling Deep Learning on Multiple In-Memory Processors	Lifan Xu, Dong Ping Zhang, Nuwan Jayasena	2015	AMD Research	EUA	Google aca-demico
22	Practical Near-Data Processing for In-memory Analytics Frameworks	Mingyu Gao, Grant Ayers, Christos Kozyrakis	2015	Stanford University	EUA	IEEE Xplore

Continuation of Table B.1

ID	Paper title	Author names	Year	Institution	Country	Search Engine
23	Catalina: In-Storage Processing Acceleration for Scalable Big Data Analytics	Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Hosein Bobarshad, Nader Bagherzadeh	2019	University of California, NDG Systems	EUA	IEEE Xplore
24	LAcc: Exploiting Lookup Table-based Fast and Accurate Vector Multiplication in DRAM-based CNN Accelerator	Quan Deng, Youtao Zhang, Minxuan Zhang, Jun Yang	2019	University of Pittsburgh	EUA	ACM
25	NIM: An HMC-Based Machine for Neuron Computation	Geraldo F. Oliveira, Paulo C. Santos, Marco A. Z. Alves, Luigi Carro	2017	Federal University of Rio Grande do Sul, Federal University of Paraná	Brazil	Springer Link
26	Exploiting Reconfigurable Vector Processing for Energy-Efficient Computation in 3D-Stacked Memories	João Paulo C. de Lima, Paulo C. Santos, Rafael F. De Moura, Marco A. Z. Alves, Antonio C. S. Beck, Luigi Carro	2019	Federal University of Rio Grande do Sul, Federal University of Paraná	Brazil	Springer Link
27	An In-DRAM Neural Network Processing Engine	Chirag Sudarshan, Jan Lappas, Muhammad Mohsin Ghaffar, Vladimir Rybalkin, Christian Weis, Matthias Jung, Norbert Wehn	2019	Technische Universität Kaiserslautern, Fraunhofer Institute for Experimental Software Engineering	Germany	IEEE Xplore

End of Table

APPENDIX C – KNN ALGORITHM WITH AVX512

```
#define AVX_SIZE 16

void euclidean_distance(char const *argv[]) {
   int i, j, k, ed_idx = 0;
   __m512 avx_tebase, avx_trbase, avx_psub, avx_pmul;

   // tr_base: training instance array; base_size: training dataset size
   read_instance(tr_base, base_size);

   // reads each test instance and partially calculates the euclidean
      distance
   for (i = 0; i < test_instances; i++) {
      ed_idx = 0;
      read_instance(te_base, training_features);
      for (j = 0; j < base_size; j += training_features) {
         for (k = 0; k < training_features/AVX_SIZE; k++) {
            avx_trbase = _mm512_load_ps(&tr_base[j + k * AVX_SIZE]);
            avx_tebase = _mm512_load_ps(&te_base[k * AVX_SIZE]);
            avx_psub = _mm512_sub_ps(avx_trbase, avx_tebase);
            avx_pmul = _mm512_mul_ps(avx_psub, avx_psub);
            e_distance[i][ed_idx] += _mm512_reduce_add_ps(avx_pmul);
         }
         ed_idx++;
      }
   }

   // after calculated the partial euclidean distance of all instances,
      calculates the square root of the distances and classify it
   for (i = 0; i < test_instances; ++i) {
      for (j = 0; j < training_instances; ++j) {
         e_distance[i][j] = sqrt(e_distance[i][j]);
      }
      get_ksmallest(e_distance[i], tr_label, knn, k_neighbors);
      classification(knn, k_neighbors);
   }
}
```

APPENDIX D – MLP HIDDEN LAYER ALGORITHM WITH AVX512

```
#define AVX_SIZE 16

float *hidden_layer_oper() {
   int i, j, k, h_idx = 0;
   __m512 avx_base, avx_weights, avx_bias, avx_phidden;

   // h_weights: sets of weights array; w_size: sets of weights size
   initialize_weights(h_weights, w_size);

   // reads each instance to multiply with the sets of weights
   for (i = 0; i < instances; i++) {
      read_instance(instance, features);
      for (j = 0; j < w_size; j += features) {
         for (k = 0; k < features/AVX_SIZE; k++) {
            avx_base = _mm512_load_ps(&instance[k * AVX_SIZE]);
            avx_weights = _mm512_load_ps(&h_weights[j + k * AVX_SIZE]);
            avx_weights = _mm512_mul_ps(avx_base, avx_weights);
            hidden_layer[h_idx] += _mm512_reduce_add_ps(avx_weights);
         }
         h_idx++;
      }
   }

   // after calculated the hidden_layer values, sum the bias and applies
      ReLU
   h_idx = 0;
   avx_bias = _mm512_set1_ps((float)1.0);
   for (i = 0; i < features/2 * instances; i += AVX_SIZE) {
      avx_phidden = _mm512_load_ps(&hidden_layer[i]);
      avx_phidden = _mm512_add_ps(avx_phidden, avx_bias);
         _mm512_store_ps(&hidden_layer[i], avx_phidden);
      for (j = i; j < AVX_SIZE; j++) {
         if (hidden_layer[j] < 0.0) {
         hidden_layer[j] = 0.0;
         }
      }
         h_idx += AVX_SIZE;
   }

   return hidden_layer;
}
```

APPENDIX E – MLP OUTPUT LAYER ALGORITHM WITH AVX512

```c
#define AVX_SIZE 16

float *output_layer_oper(float *hidden_layer) {
   int i, j, k, o_idx = 0;
   __m512 avx_hidden, avx_oweights, avx_output, avx_bias, avx_mul;

   // o_weights: sets of weights array; ow_size: sets of weights size
   initialize_weights(o_weights, ow_size);

   // multiply the hidden layer activation values with the weights
   for (i = 0; i < hidden_size; i += features/2) {
      for (j = i * output_size; j < (i * output_size) + features; j +=
          features/2) {
         for (k = 0; k < (features/2)/AVX_SIZE; k++) {
            avx_hidden = _mm512_load_ps(&hidden_layer[i + k * AVX_SIZE]);
            avx_oweights = _mm512_load_ps(&o_weights[k * AVX_SIZE]);
            avx_oweights = _mm512_mul_ps(avx_hidden, avx_oweights);
            output_layer[o_idx] += _mm512_reduce_add_ps(avx_oweights);
         }
         o_idx++;
      }
   }

   // after calculated the output_layer values, sum to the bias
   avx_bias = _mm512_set1_ps((float)1.0);
   for (i = 0; i < output_size * instances; i += AVX_SIZE) {
      avx_output = _mm512_load_ps(&output_layer[i]);
      avx_output = _mm512_add_ps(avx_output, avx_bias);
      _mm512_store_ps(&output_layer[i], avx_output);
   }

   return output_layer;
}
```

APPENDIX F – CONVOLUTION ALGORITHM WITH AVX512

```
#define AVX_SIZE 16

int main(int argc, char const *argv[]) {
   __m512 dim_a1, dim_a2, dim_a3, dim_a4, dim_a5, dim_a6, dim_a7, dim_a8,
      dim_a9 dim_b;

   // initialize vectors A and B
   __m512 vec_a = _mm512_set1_ps((float) 1.0);
   for (int i = 0; i < v_size; i += AVX_SIZE)
      _mm512_store_ps(&data_a[i], vec_a);

   vec_a = _mm512_set1_ps((float) 0.0);
   for (int i = 0; i < v_size; i += AVX_SIZE)
      _mm512_store_ps(&data_b[i], vec_a);

   // convolution operation with 9 cells
   __m512 mul = _mm512_set1_ps((float)2.0);
   for (int i = dim_size; i+dim_size+AVX_SIZE < v_size; i += AVX_SIZE) {
      dim_a1 = _mm512_loadu_ps(&data_a[i - dim_size - 1]);
      dim_a2 = _mm512_load_ps(&data_a[i - dim_size]);
      dim_a3 = _mm512_loadu_ps(&data_a[i - dim_size + 1]);
      dim_a4 = _mm512_loadu_ps(&data_a[i - 1]);
      dim_a5 = _mm512_load_ps(&data_a[i]);
      dim_a6 = _mm512_loadu_ps(&data_a[i + 1]);
      dim_a7 = _mm512_loadu(&data_a[i + dim_size - 1]);
      dim_a8 = _mm512_load_ps(&data_a[i + dim_size]);
      dim_a9 = _mm512_loadu(&data_a[i + dim_size + 1]);

      dim_b = _mm512_add_ps(dim_a1, dim_a2);
      dim_b = _mm512_add_ps(dim_b, dim_a3);
      dim_b = _mm512_add_ps(dim_b, dim_a4);
      dim_b = _mm512_add_ps(dim_b, dim_a5);
      dim_b = _mm512_add_ps(dim_b, dim_a6);
      dim_b = _mm512_add_ps(dim_b, dim_a7);
      dim_b = _mm512_add_ps(dim_b, dim_a8);
      dim_b = _mm512_add_ps(dim_b, dim_a9);
      dim_b = _mm512_mul_ps(dim_b, mul);
      _mm512_stream_ps (&data_b[i], dim_b);
   }
   return 0;
}
```

APPENDIX G – KNN ALGORITHM WITH 8KB VIMA VECTOR

This case refers to the case with instances smaller than the VIMA vector.

```
void euclidean_distance(char const *argv[]) {
    int32_t i, j, k, ed_idx = 0;

    // mask is defined to isolate each training and test instance
    for (i = 0; i < training_features; ++i) {
        mask[i] = 1.0;
    }

    // read test instances
    for (i = 0; i < test_instances; i += n_instances) {
        _vim2K_fmovs(0.5, te_base);

        // apply the mask over training instances
        for (j = 0; j < training_instances; ++j) {
            _vim2K_fmuls(&tr_base[j* training_features], mask, temp_train);

            // apply the mask over test instances
            for (k = 0; k < n_instances; ++k) {
                _vim2K_fmuls(&te_base[k * training_features], mask, temp_test);
                _vim2K_fsubs(temp_train, temp_test, temp_test);
                _vim2K_fmuls(temp_test, temp_test, temp_test);
                _vim2K_fcums(temp_test, &e_distance[ed_idx++]);
            }
        }
    }

    // square root operation with x86 over partial Euclidean Distance array
    for (i = 0; i < test_instances * training_instances; ++i) {
        e_distance[i] = sqrt(e_distance[i]);
    }

    // classify an instance
    for (i = 0; i < test_instances * training_instances; i +=
        training_instances) {
        get_ksmallest(&e_distance[i], tr_label, knn, k_neighbors);
        classification(knn, k_neighbors);
    }
}
```

APPENDIX H – KNN ALGORITHM WITH VIMA 8KB VECTOR

This case refers to the case with instances greater or equal to the VIMA vector.

```
#define VSIZE 2048

void euclidean_distance(char const *argv[]) {
    int32_t i, j, k, ed_idx = 0;

    // for instances greater or equal to VIMA vector size a mask is not
        needed
    for (i = 0; i < test_instances; ++i) {

        // read test instance
        for (j = 0; j < n_vectors; ++j) {
            _vim2K_fmovs(0.5, &te_base[j * VSIZE]);
        }

        // partial Euclidean Distance between training and test instances
        for (j = 0; j < training_instances; ++j) {
            for (k = 0; k < n_vectors; ++k) {
                _vim2K_fsubs(&tr_base[(j * VSIZE * n_vectors) + (k * VSIZE)], &
                    te_base[k * SIZE], temp_test);
                _vim2K_fmuls(temp_test, temp_test, temp_test);
                _vim2K_fcums(temp_test, &partial_sum[k]);
            }
            _vim2K_fcums(partial_sum, &e_distance[ed_idx++]);
        }
    }

    // square root operation with x86 over partial Euclidean Distance array
    for (i = 0; i < test_instances * training_instances; ++i) {
        e_distance[i] = sqrt(e_distance[i]);
    }

    // classify an instance
    for (i = 0; i < test_instances * training_instances; i +=
        training_instances) {
        get_ksmallest(&e_distance[i], tr_label, knn, k_neighbors);
        classification(knn, k_neighbors);
    }
}
```

APPENDIX I – MLP HIDDEN LAYER ALGORITHM WITH VIMA 8KB VECTOR

This case refers to the case with instances smaller than the VIMA vector.

```
__v32f *hidden_layer_oper() {

   // defines the set of weights
   for (i = 0; i < weight_size; i += VSIZE) {
      _vim2K_fmovs(1.0, weights);
   }

   // defines mask to isolate each instance
   for (i = 0; i < features; ++i) {
      mask[i] = 1.0;
   }

   // reads the instances and apply the mask over the instances
   for (i = 0; i < instances; i += instance_size) {
      _vim2K_fmovs(1.0, instance_vector);
      for (j = 0; j < VSIZE; j += features) {
         _vim2K_fmuls(&instance_vector[j], mask, temp_instance);

         // calculates partial hidden layer activation values
         for (k = 0; k < features/2; ++k) {
            _vim2K_fmuls(&weights[(k * features)], mask, temp_weights);
            _vim2K_fmuls(temp_instance, temp_weights, temp_weights);
            _vim2K_fcums(temp_weights, &hidden_layer[h_idx++]);
         }
      }
   }

   // adds the bias
   _vim2K_fmovs(1.0, bias);
   for (i = 0; i < hidden_size; i += VSIZE) {
      _vim2K_fadds(&hidden_layer[i], bias, &hidden_layer[i]);
   }

   // applies ReLU
   _vim2K_fmovs(0.0, relu);
   for (i = 0; i < hidden_size; i += VSIZE) {
      _vim2K_fmaxs(&hidden_layer[i], relu, &hidden_layer[i]);
   }
}
```

APPENDIX J – MLP HIDDEN LAYER ALGORITHM WITH VIMA 8KB VECTOR

This case refers to the case with instances greater or equal to the VIMA vector.

```
#define VSIZE 2048

__v32f *hidden_layer_oper() {

    // defines the set of weights
    for (i = 0; i < weight_size; i += VSIZE) {
        _vim2K_fmovs(1.0, weights);
    }

    // reads an instance
    for (i = 0; i < instances; ++i) {
        for (j = 0; j < n_vectors; ++j) {
            _vim2K_fmovs(1.0, &instance_vector[j * VSIZE]);
        }

        // calculates partial hidden layer activation values
        for (j = 0; j < features/2; ++j) {
            for (k = 0; k < n_vectors; ++k) {
                _vim2K_fmuls(&instance_vector[k * VSIZE], &weights[(j *
                    features) + (k * VSIZE)], temp_weights);
                _vim2K_fcums(temp_weights, &p_sum[k]);
            }
            _vim2K_fcums(p_sum, &hidden_layer[h_idx++]);
        }
    }

    // adds the bias
    _vim2K_fmovs(1.0, bias);
    for (i = 0; i < hidden_size; i += VSIZE) {
        _vim2K_fadds(&hidden_layer[i], bias, &hidden_layer[i]);
    }

    // apply ReLU with x86 operations
    for (i = 0; i < hidden_size; ++i) {
        if (hidden_layer[i] < 0.0)
        hidden_layer[i] = 0.0;
    }
}
```

APPENDIX K – CONVOLUTION ALGORITHM WITH VIMA 8KB VECTOR

```
1  #define VSIZE 2048
2
3  int main() {
4
5     // Initializes vectors A and B
6     for (i = 0; i < v_size; i += VSIZE) {
7        _vim2K_fmovs(1, &vector_a[i]);
8        _vim2K_fmovs(0, &vector_b[i]);
9        _vim2K_fmovs(1, &mul[i]);
10    }
11
12    // convolution operation with 9 cells
13    for (int i = dim_size; i + dim_size + VSIZE < v_size; i += VSIZE) {
14       _vim2K_fadds(&vector_b[i], &vector_a[i-dim_size-1], &vector_b[i]);
15       _vim2K_fadds(&vector_b[i], &vector_a[i-dim_size], &vector_b[i]);
16       _vim2K_fadds(&vector_b[i], &vector_a[i-dim_size+1], &vector_b[i]);
17       _vim2K_fadds(&vector_b[i], &vector_a[i-1], &vector_b[i]);
18       _vim2K_fadds(&vector_b[i], &vector_a[i], &vector_b[i]);
19       _vim2K_fadds(&vector_b[i], &vector_a[i+1], &vector_b[i]);
20       _vim2K_fadds(&vector_b[i], &vector_a[i+dim_size-1], &vector_b[i]);
21       _vim2K_fadds(&vector_b[i], &vector_a[i+dim_size], &vector_b[i]);
22       _vim2K_fadds(&vector_b[i], &vector_a[i+dim_size+1], &vector_b[i]);
23       _vim2K_fmuls(&vector_b[i], &mul[i], &vector_b[i]);
24    }
25 }
```