MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO - UFERSA
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO - PROPPG
Av. Francisco Mota, 572 – C. Postal 137 – Bairro Pres. Costa e Silva – Mossoró – RN – CEP: 59.625-900 - Tel.: (84)3317-8296/8295 – E.mail: proppg@ufersa.edu.br

REQUERIMENTO E ANEXOS PARA RENOVAÇÃO DE AFASTAMENTOS DE SERVIDORES DOCENTES DA UFERSA PARA QUALIFICAÇÃO EM INSTITUIÇÕES NACIONAIS OU ESTRANGEIRAS EM NÍVEL DE PÓS-GRADUAÇÃO *STRICTO SENSU*

1. PREENCHIDO PELO REQUERENTE

Nome: SAIRO RAONÍ DOS SANTOS
Identidade: 2459204 **Órgão Emissor:** SSP **UF:** RN **Data de emissão:** 07/01/2016
CPF: 014.332.924-32 **Data de Nascimento:** 01/01/1990 **Tel.:** (84) 99830-1664
E-mail: sairo.santos@ufersa.edu.br **Departamento/Setor:** DCETI
Tipo de Afastamento: Integral: (x) **Parcial:** ( )
Tempo de Serviço Averbado para Aposentadoria: ( ) **Anos**
Início de Exercício no Cargo: 06/05/2013 **Total:** 7 anos e 10 meses

2. PREENCHIDO PELO REQUERENTE

CURSO: Programa de Pós-graduação em Informática
Nível: Doutorado
Área de concentração: Ciência da Computação
Liberação inicial: Início 09/07/2018 **Término:** 08/07/2019
Período solicitado para (renovação): Início 09/07/2021 **Término:** 08/07/2022
Previsão para término do curso: Início: 09/07/2018 **Término:** 08/07/2022

ANEXAR

I. Lista de verificação própria disponibilizada pela PROPPG (**Check-List**); ***(Anexo I)***
II – Justificativa de seu requerimento; ***(Anexo II)***
III- Relatório de atividades acadêmicas (Anexo III) (quando se tratar do relatório referente ao 3º semestre (mestrado) e 5º semestre (doutorado), deverá ser acompanhado do **projeto de dissertação/Tese)**
IV- Relatório de avaliação de desempenho, feito pelo/a orientador/a (Anexo IV)
V - Declaração de matrícula (Local da pós-graduação) (Anexo V)
VI- Histórico Escolar (Anexo VII) (Disponível na Página da PROPPG**)**
VII- Termo de Compromisso dos docentes que assumirão os componentes curriculares do docente afastado, durante o período de renovação do afastamento, restrito aos casos de indisponibilidade de vaga para contratação de professor substituto; ***(Anexo VII)***
VIII – Termo de Compromisso, devidamente preenchido e assinado com testemunhas; ***(Anexo VIII)***
IX - Parecer da chefia imediata (Departamento acadêmico de lotação do requerente); ***(Anexo IX)***
X - Parecer do Conselho do Centro ao qual o requerente faz parte. ***(Anexo X).***

PROPPG

Data: 11/04/2021

SAIRO RAONI DOS SANTOS:01433292432

Assinado de forma digital por SAIRO RAONI DOS SANTOS:01433292432
Dados: 2021.04.12 08:36:15 -03'00'

Assinatura do requerente

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO - UFERSA
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO - PROPPG
Av. Francisco Mota, 572 – C. Postal 137 – Bairro Pres. Costa e Silva – Mossoró – RN – CEP: 59.625-900 - Tel.: (84)3317-8296/8295 – E.mail: proppg@ufersa.edu.br

Check-List – Renovação de Afastamento para qualificação

Nome do solicitante: Sairo Raoní dos Santos	
Local da Qualificação*:*	
☐ No País X ☐ No exterior	
Período solicitado para renovação do afastamento: 09/07/2021 a 08/07/2022	
Documentos Anexados – Processo de Renovação:	**Número da página**
I. Lista de verificação própria disponibilizada pela PROPPG (Check-List); ***(Anexo I)***	
II. Justificativa de seu requerimento; (***Anexo II***)	
III. Relatório de atividades acadêmicas (***Anexo III***)	
IV. Relatório de avaliação de desempenho, feito pelo orientador (***Anexo IV***)	
V. Declaração de Matrícula ***(Anexo V)***	
VI. Histórico Escolar – Atualizado ***(Anexo VI)***	
VII – Termo de Compromisso, devidamente preenchido e assinado com testemunhas; (Anexo VIII)	
VIII. Documentação que formalize a substituição do(a) interessado: ***(Anexo VIII)*** ☐ Utilização de vaga ou disponibilidade de professor substituto a ser contratado(a) ☐ Termo de Compromisso dos docentes que assumirão as disciplinas	
IX**.** Parecer da chefia imediata (Departamento acadêmico de lotação do requerente); ***(Anexo IX)***	
X. Parecer do Conselho do Centro ao qual o requerente faz parte. ***(Anexo X).***	

PROPPG

JUSTIFICATIVA PARA O AFASTAMENTO

Eu, **Sairo Raoní dos Santos**, portador do CPF nº 01433292432, RG nº 2459204, matrícula SIAPE nº 2975474, professor do curso de Bacharelado em Sistemas de Informação da Universidade Federal Rural do Semi-árido – UFERSA/Campus Angicos, venho por meio deste solicitar a renovação de afastamento integral de minhas atividades no período de 09 de julho de 2018 a 08 de julho de 2022 para a realização do curso de doutorado no Programa de Pós-graduação em Informática da Universidade Federal do Paraná (UFPR). Ressalto que estou classificado no Plano Anual de Qualificação e Formação Docente 2017/2018 do campus Angicos.

O Programa de Pós-graduação em Informática é oferecido pelo Departamento de Informática da UFPR e recebeu conceito 5 na sua última avaliação pela CAPES. Oferece desde 2009 o curso de Doutorado em Informática, primeiro doutorado público na área de Ciência da Computação no Paraná, atualmente com mais de 60 teses de doutorado defendidas.

O programa tem como objetivo proporcionar a seus alunos uma formação de forte conteúdo conceitual, sem desprezar aspectos práticos da informática. O corpo discente é motivado para o trabalho de pesquisa, no qual a visão crítica predomina sobre o aprendizado de técnicas e ferramentas.

Fui aprovado em março de 2018 no programa na área de Redes e Sistemas Distribuídos. Considerando os benefícios para a instituição e para a carreira como professor e pesquisador, enfatizo a importância do afastamento para a realização do doutorado.

PROPPG

Data: 11 de abril de 2021

SAIRO RAONI DOS SANTOS:01433292432

Assinado de forma digital por SAIRO RAONI DOS SANTOS:01433292432
Dados: 2021.04.12 08:36:43 -03'00'

Assinatura do requerente

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO - UFERSA
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO - PROPPG
Av. Francisco Mota, 572 – C. Postal 137 – Bairro Pres. Costa e Silva – Mossoró – RN – CEP: 59.625-900 - Tel.: (84)3317-8296/8295 – E.mail: proppg@ufersa.edu.br

RELATÓRIO DE ATIVIDADES ACADÊMICAS

Dediquei-me no último ano à escrita e subsequente defesa da minha qualificação e à escrita e subsequente submissão de artigos científicos referentes à pesquisa que venho desenvolvendo. Seguem o texto da qualificação, de título *Near-Data Bloom Filters for Efficient Data Fetch*, e o artigo de título *Machine Learning Migration for Efficient Near-Data Processing* publicado no evento PDP (*Parallel, Distributed and Network-Based Processing*).

Data: 11 de abril de 2021

SAIRO RAONI DOS SANTOS:01433292432

Assinado de forma digital por SAIRO RAONI DOS SANTOS:01433292432
Dados: 2021.04.12 08:37:10 -03'00'

Assinatura do requerente

Prof. Marco A. Zanata Alves
Dep. de Informática
Mat.: 205129 - UFPR

Assinatura do Orientador

PROPPG

Machine Learning Migration for Efficient Near-Data Processing

Aline S. Cordeiro[†] Sairo R. dos Santos[†‡] Francis B. Moreira[†] Paulo C. Santos[§] Luigi Carro[§] Marco A. Z. Alves[†]
[†]Department of Informatics – Federal University of Paraná – Curitiba, Brazil
[‡]Department of Exact Sciences and Information Technology – Federal Rural University of Semi-arid – Angicos, Brazil
[§]Informatics Institute – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
Email:[†]{ascordeiro, fbm, mazalves}@inf.ufpr.br [‡]{sairo.santos@ufersa.edu.br} [§]{pcssjunior, carro}@inf.ufrgs.br

***Abstract*—Machine Learning (ML) rises as a highly useful tool to analyze the vast amount of data generated in every field of science nowadays. 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 increasing data by drastically reducing the required amount of data movement. For NDP, we see three main approaches, Application-Specific Integrated Circuits (ASICs), full Central Processing Units (CPUs) and Graphics Processing Units (GPUs), or vector units integration. However, previous work considered only ASICs, CPUs and GPUs when executing ML algorithms inside the memory. In this paper, we present an approach to execute ML algorithms near-data, using a general-purpose vector architecture and applying near-data parallelism to kernels from KNN, MLP, and CNN algorithms. To facilitate this process, we also present an NDP intrinsics library to ease the evaluation and debugging tasks. Our results show speedups up to $10\times$ for KNN, $11\times$ for MLP, and $3\times$ for convolution when processing near-data compared to a high-performance x86 baseline.**

***Index Terms*—Near-Data Processing; Vector Processing; Machine Learning.**

I. INTRODUCTION

In the last years, Machine Learning (ML) has gained popularity to analyze the massive amounts of data generated by digital systems' growth use [1]–[5]. Simultaneously, general-purpose computers and their ever-increasing performance present severe bottlenecks in terms of the execution time of ML algorithms when dealing with real-world size problems [6]. In order to mitigate performance problems, ML experts are using accelerators such as Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) [7]–[9]. However, the data movement between accelerator's integrated memory (GDDR-x, HBM, or HMC) and processor is still a bottleneck, also known as memory-wall [7]. The memory-wall limitation is inherent to contemporary processor designs, and although cache hierarchy can mitigate the performance drawbacks, in terms of energy and latency it is not sufficient [10]–[12].

Data movement consumes as high as 60% of the total system energy [6]. Here, Near-Data Processing (NDP) has emerged as a solution for the memory-wall problem, with the idea of integrating processor and memory in the same chip [13], [14]. The most common NDP proposals rely on ASIC or full Central Processing Units (CPUs) and GPUs [15]–[17]. Nevertheless, prominent designs, based on simple near-data vector units [18]–[21], enable the highest energy efficiency while meeting the required constraints regarding the area and power [22]. Therefore, our case study is inspired by the HMC Instruction Vector Extensions (HIVE) [18] to provide a programming and simulation environment for NDP.

In this paper, we present the benefits of migrating three well-known ML kernels, namely K-Nearest Neighbors (KNN), Multi-layer Perceptron (MLP), and Convolutional Neural Network (CNN), to a NDP design capable of large-vector operations which is named Vector-In-Memory Architecture (VIMA). By adopting VIMA we are greatly reducing data movement between host processors and main memory, hence increasing overall efficiency and performance. To allow this migration, we also developed Intrinsics-VIMA, a vector-designed C/C++ library extension [23]. Intrinsics-VIMA facilitates the writing of codes for VIMA and similar Processing-In-Memory (PIM) architectures, enabling the simulation and evaluation of new algorithms with reduced programming effort. Our main contributions are the following:

- We extend and use an NDP intrinsics library that supports validation of NDP architectures based on large vectors.
- We provide insights and show benefits on migrating ML algorithms to a vector-based NDP architecture.

Most ML algorithms are split into train and inference phases, two computation-intensive tasks. The training is performed once and relies on latency to execute many operations over a massive set of training instances to define the model parameters. The inference is performed multiple times by multiple products, and it relies on high throughput to classify a stream of instances, representing real-world applications. In this paper, we focus only on the inference phase.

Comparing the x86-only approach to the NDP execution we show improvements on execution time up to $10\times$ for KNN, $11\times$ for MLP, and $3\times$ for convolution. Additionally, we reduce energy consumption by up to $7\times$ for KNN, $\sim 8\times$ for MLP and $2\times$ for convolution.

This work was partially supported by the Serrapilheira Institute (grant number Serra-1709-16621), CAPES and CNPq (Brazilian Government).

II. RELATED WORK

In this section, we discuss related work on NDP for ML execution. We begin by describing efforts that rely on using full cores, such as RISC-V, ARM, and Accelerated Processing Units (APUs) attached to a 3D-stacked architecture to enhance performance. NeuroStream is a NDP platform that runs Deep Neural Networks (DNNs) with large inputs and arbitrary filter sizes [24]. Based on NeuroStream, Network Training Accelerator (NTX) implements an acceleration engine that trains state-of-the-art Deep Convolutional Neural Networks (DCNNs) [25]. Both implement a module composed of RISC-V cores with local cache, Direct Memory Access (DMA), and specific cores. The modules connect to the crossbar switch of every 3D-stacked memory, enabling the execution of vector instructions. Tesseract accelerates large-scale graph processing using an Hybrid Memory Cube (HMC) module integrated to a single-issue in-order ARM core [16]. Another NDP architecture was used for in-memory analytics frameworks [26], where the authors employ a set of ARM cores combined with a Translation Look-aside Buffers (TLBs) and virtual memory that communicate with each other through a vault router. The Millipede is a NDP architecture for Big data Machine Learning Analytics (BMLA), that implements its processors in the logic layer of 3D-stacked memories. These processors have a local memory, register file, pipeline, cache, and prefetcher buffers [27]. Another possible approach is to implement programmable ARM-based cores in the HMC logic layer [28] so that some functions can be offloaded to these cores. The VIMA vector module also attaches to the crossbar switch but has lower complexity and cost as it requires fewer components to improve system performance for a ML application.

Another proposal analyzed aspects of a CNN to develop a PIM architecture where simple cores are attached to every vault, and each core has a data controller to allow communication [29]. TETRIS and NeuralHMC are 3D-stacked Neural Network (NN) accelerators [15], [30]. Both connect hundreds of Processing Elements (PEs) with Network-on-Chip (NoC) technology. VIMA also has lower complexity and cost than these, as it does not rely on communication between vaults or cores to achieve high performance or parallelism.

MAssively Parallel Learning/Classification Engine (MAPLE) [31] uses multi-core near-data for parallel learning and classification algorithms and a tool that automatically maps application kernels to the accelerator hardware. They implemented an architecture with a set of cores to solve MapReduce operations. MAPLE uses these processing cores to achieve parallelism, and two separate modules are applied to solve the entire operation. The cores include processing elements like registers, selectors, a vector Functional Unit (FU), and local storage. Xu et al.'s proposal [32] focuses on parallelizing CNNs on a system with multiple NDP devices. A host CPU is connected to a 3D-stacked memory and both host and logic layer are APUs, which consists of CPU and GPU cores on the same silicon die. VIMA, on the other hand, uses a straightforward module, enabling vector operation in an energy-efficient way.

Another approach is the addition of reconfigurable accelerators to the logic layer of 3D-memories. Oliveira et al. [20] describe Neuron In-Memory (NIM), a module compound by a register bank, complex FUs, and a sequencer that simulates biologically meaningful NN of considerable sizes. We also observed proposals that dynamically adjusts the number of active FUs on demand [33]. These related proposals require adding one module per vault, making them more expensive than VIMA, which is attached only to the crossbar switch and allows communication to every vault.

Finally, some proposals consider a conventional Dynamic Random Access Memory (DRAM) device with elementary logic or boolean circuit into DRAM cells (so-called Processing In-Memory), which is not an expensive task. However, compared to VIMA, this solution is a complex and error-prone task to the programmer. Moreover, the set of implementable instructions is limited [34]–[39].

Table I summarizes the related work regarding NDP and PIM applied to ML algorithms.

TABLE I: Summary of correlated papers characteristics.

Paper	General/Specific Purpose	Vector/ Scalar	Near-/In-memory	# Full Cores
[27], [31], [33]	General	Vector	Near-memory	N
[20], [36]	General	Vector	In-memory	1
[16], [26], [32]	General	Scalar	Near-memory	N
[37]	Specific	Vector	In-memory	1
[15], [24]	Specific	Vector	Near-memory	N
[34], [35], [38]	Specific	Scalar	In-memory	1
[25], [28]	Specific	Scalar	Near-memory	N
[29], [30]	Specific	Scalar	Near-memory	0
Our Proposal	**General**	**Vector**	**Near-memory**	**0**

III. BACKGROUND ON NEAR DATA PROCESSING

Near-Data Processing (NDP) dates back to the 1990s [14], [40], 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. 3D-stacked memories are generally compound of multiple stacked layers (e.g., eight layers) of DRAMs plus a logic layer on the base. This logic layer enables the integration of a processing logic element near the memory banks. The DRAM layers are usually logically partitioned (e.g., in up to 32 vaults), where each partition has many independent DRAM banks (from all the eight layers). These logical partitions distributed among DRAM layers are connected through Through-Silicon Vias (TSVs) [41]. Compared to typical Double Data Rate (DDR) memories, 3D memories can achieve higher bandwidth and better energy efficiency [42], while reaching up to 320 GB/s [43], [44].

NDP systems can be implemented due to 3D integration technology by adding processing capabilities within the logic layer. Thereby, NDP can mitigate data movement between memory and processor because it enables processing in the same chip where data is stored. NDP architecture improves performance and energy efficiency as it grants high parallelism and high bandwidth [45]–[47], ensuring low average latency even when there is high pressure in memory. Therefore, such architectures benefit streaming and parallel applications, with coalescent memory access patterns and low data reuse.

In this paper, as a target NDP architecture, we focus on a model that provides general-purpose processing (e.g., in contrast to ASIC) and does not require a full processor integration near data. For this, we adopted the HIVE [18] architecture. It allows the execution of large vector instructions that obtain data from the independent memory vaults inside a 3D-memory in a parallel fashion. Besides, it includes vector extensions to the processor Instruction Set Architecture (ISA) to control the near-data vector units, not requiring any processor front-end to be implemented inside the memory.

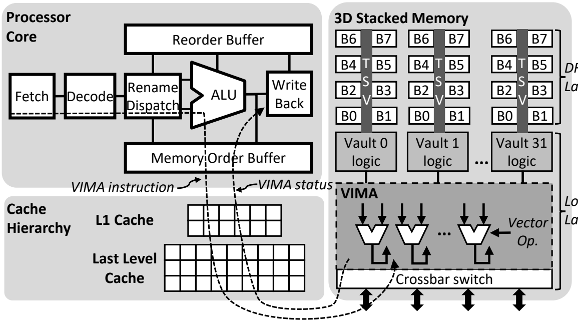


Fig. 1: 3D-stacked memory module with the VIMA architecture.

For our experiments, we used VIMA, a modified version of HIVE. The main difference is that VIMA replaces the register bank (from HIVE) with a same-sized (i.e., 64 KB) data cache memory, which maintains high performance while providing transparency and high flexibility for programmers. Both HIVE and VIMA support all ARM NEON Integer and Floating-point (FP) instructions and operate over vectors of 8 KB of data, which fetch data over the 32 channels (vaults) in parallel. Figure 1 present an overview of the NDP considered in this paper. For more details please refer to HIVE's paper [18].

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. Thus, in the remainder of this paper, we migrate three well-known ML classification algorithms to exploit this emerging architecture.

IV. PROPOSED INTRINSICS-VIMA LIBRARY

In this section, we present the Intrinsics-VIMA, a library we develop intending to facilitate the development of programs for NDP architectures using *C/C++* language.

Intrinsics-VIMA supports trace generation for simulation and allows vector operation with vectors of 8 KB formed by multiple integers, single- or double-precision floating-point elements. This library is based on Intel and ARM Intrinsics [48], a Single Instruction Multiple Data (SIMD) library that embeds its internal assembly code directly in the compiler to optimize execution [49].

The main idea for Intrinsics-VIMA is to provide vector extensions in the ISA. Once *C/C++* code is prepared using Intrinsics-VIMA, it can be debugged and executed on any architecture. However, to evaluate new NDP architectures, we use a trace generator to transform each intrinsic call into a specific NDP instruction supported by the simulator. Thus, it is possible to write code for non-existing architectures and ensure its correctness [23].

Code 1: Intrinsics-VIMA routine call for vector sum.

```
uint32_t vima_size = 2048;

// Allocate the vectors A, B (sources) and C (result)
__v32f *A = (__v32f*)malloc(sizeof(__v32f)* vima_size* x);
__v32f *B = (__v32f*)malloc(sizeof(__v32f)* vima_size* x);
__v32f *C = (__v32f*)malloc(sizeof(__v32f)* vima_size* x);

// Initialize the memory location
<...>

// Perform the vector sum: C[i] = A[i] + B[i]
for (int i = 0; i < vima_size * x; i += vima_size) {
    _vim2K_fadds(&A[i], &B[i], &C[i]);
}
```

Based on this open-source Intrinsics library for NDP [23], we developed Intrinsics-VIMA, which is the first library to implement vector instructions for an NDP architecture. VIMA instructions operate over 8 KB and allow (un)signed integer and floating-point single- and double-precision operations. Thus, we consider vectors with 1024 or 2048 elements, for 8 B or 4 B elements, respectively. To use intrinsics-VIMA, we must allocate vectors with sizes multiple of 1024 or 2048, so we can iterate on these vectors with this stride length. Code 1 present the implementation of a vector sum example using Intrinsics-VIMA. Code 2 show the implementation of one of our Intrinsics-VIMA routine. Previous work only considered scalar processing near-data (based on HMC ISA proposal) [23], and now we can evaluate vector operands used on VIMA, originally inspired on the NEON ISA.

Code 2: Intrinsics-VIMA routine example.

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

V. MIGRATING MACHINE LEARNING USING INTRINSICS-VIMA

ML is a sub-field of Artificial Intelligence (AI), and its algorithms compute and analyze datasets to recognize patterns in data and classify or predict them. Commonly we split ML algorithms into training and inference phases. Considering supervised algorithms, developers perform the training, and once it is validated and ready, these trained values are embedded into multiple systems. It can be executed in a set of different devices, even in embedded systems with limited hardware resources [50]–[52]

Both phases are computation-intensive tasks and may present different challenges. The training depends on massive operations over a massive set of instances during multiple epochs to define the model parameters. Meanwhile, inference relies on high throughput to classify a stream of instances, representing real-time applications. Therefore, for simplicity, we choose to focus on this inference phase only in this paper.

In the following subsection, we describe the implementation of three algorithms widely adopted in ML, also showing the method to vectorize each of them. We choose a convolution kernel (commonly used in CNNs), MLP and KNN algorithms.

Besides, we use VIMA vectors of 8 KB, allowing us to operate over 2048 single-precision values with a single instruction. Although HIVE and VIMA instructions operates over 8 KB vectors. The physical implementation of these architectures can use less vector units in a pipeline manner to still provide high performance while low area usage [18].

A. Convolution

We start explaining the convolution code due to its simplicity, making it easy to understand the vector process. Convolution codes are a class of algorithms that has numerous applications in science. They compute values based on a fixed pattern involving each element of an array and several neighbors on a 2D or 3D arrangement [53]. Two of the most common convolution patterns are the Von Neumann neighborhood and the Moore neighborhood patterns. The Von Neumann pattern includes the four neighbors in the cardinal directions of an element. 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 [53].

Code 3: Von Neumann convolution code in C.

```
for (int i = ColSize; i < max_elem; i++) {
    VecB[i] = VecA[i]; // Center Elem.
    VecB[i] = VecB[i] + VecA[i - ColSize]; // Upper Elem.
    VecB[i] = VecB[i] + VecA[i + ColSize]; // Lower Elem.
    VecB[i] = VecB[i] + VecA[i - 1]; //Left Elem.
    VecB[i] = VecB[i] + VecA[i + 1]; //Right Elem.
    VecB[i] = VecB[i] * constK;
}
```

For the implementation of a naive convolution code using VIMA, we adopted the Von Neumann pattern with a range equals to 1, as shown in dark gray in Figure 2. The algorithm sums all five elements in the convolution, then multiplies the result by a constant and stores the result in a different matrix. Code 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.

Figure 2 illustrates the convolution. For every loop, elements from three consecutive lines of the matrix, as pictured in dark gray, are loaded into VIMA vectors and operated over. Code 4 shows the implementation using Intrinsics-VIMA. Our implementation is considering a convolution that eliminates the matrix borders during execution.

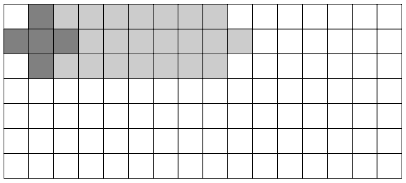


Fig. 2: Convolution pattern used for VIMA.

Code 4: Von Neumann convolution using Intrinsics-VIMA.

```
for (int i = ColSize; i < max_elem; i += vec_size) {
    _vim2K_fmovs(&VecA[i], &VecB[i]);
    _vim2K_fadds(&VecB[i], &VecA[i-ColSize], &VecB[i]);
    _vim2K_fadds(&VecB[i], &VecA[i+ColSize], &VecB[i]);
    _vim2K_fadds(&VecB[i], &VecA[i+1], &VecB[i]);
    _vim2K_fadds(&VecB[i], &VecA[i-1], &VecB[i]);
    _vim2K_fmuls(&VecB[i], &VconstK[i], &VecB[i]);
}
```

B. K-nearest Neighbors

KNN is an instance-based classifier. It searches for the *k* minimal distances between training and test points in an n-dimensional space. Here we use the Euclidean method to calculate the instances' distances. An n-dimensional array of features represents each instance. Each array position corresponds to a different feature, which also corresponds to a weight. The higher the value, the heavier it is [54].

In the KNN algorithm, we must access the training data in memory to classify every test instance. 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 3. Meanwhile, if the instance size is equal or larger than a VIMA vector, it will occupy at least one VIMA vector.

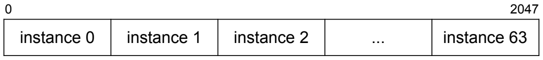


Fig. 3: E.g., full utilization of a VIMA vector with training and test instances. Here, we could allocate 64 instances with 32 features inside the vector of 8 KB.

We used an input set labeled with two classes: 0 (negative) and 1 (positive). We load the labels of the training instances into separated vectors. As we load the full training set in memory, a vector with a size multiple of the VIMA vector size must allocate all the training labels. Thus, if we store a set of 8192 training instances using 4× VIMA vectors of 8 KB, each with 2048 positions to store the 8192 labels, shown in Figure 4.

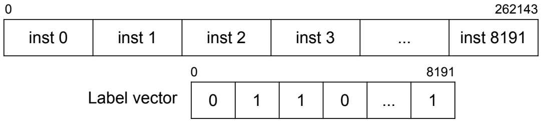


Fig. 4: VIMA vectors with training instances with 32 features and the respective labels.

With all training instances stored in memory, the next step is to calculate the Euclidean distance method, represented by the following simplified function:

$$d \equiv \sqrt{\sum_{i=1}^{n}(te(x_i) - tr(x_i))^2}$$

Where *tr* refers to the training instance and *te* to the test instance. We use the following Intrinsics-VIMA 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 results to find out the distance between these instances and finally calculates the square root of this value. Fortunately, we can vectorize most of these operations with Intrinsics-VIMA.

Although a VIMA vector can receive more than one instance, depending on the number of features, we choose to work with a single instance at a time. To do so, we apply a mask in training and test vectors to obtain just a single instance. For instance, considering 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 5. If the instances size are equal or greater than the VIMA vector, this transformation will not be necessary. Isolating one instance per VIMA vector enables executing all the operations mentioned above (subtraction, multiplication and accumulated sum) in a simpler way with better data reuse.

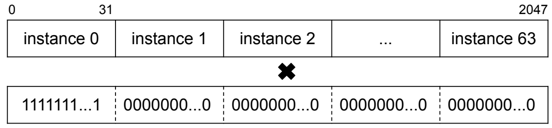


Fig. 5: Operation to apply a mask over a VIMA vector of 8 KB with instances representing 32 features.

We store the accumulated sums between each test instance and the set of training instances (calculated with VIMA routines) in a different vector in memory. Afterward, the x86 square root instruction will be applied, resulting in the Euclidean Distances. One vector for each test instance will store several Euclidean Distances. Each vector has size equals to the number of training instances. Finally, to classify an instance, all its distances are paired with the label vector to find the *k* lowest distances. In this phase, we are interested 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.

C. Multi-layer Perceptron

The MLP algorithm is a supervised learning technique that provides a practical method for learning from given examples. It is an Artificial Neural Network (ANN) that consists of one input layer, at least one hidden layer, and one output layer. Each layer is formed by neurons that apply a series of non-linear transformations on features data to classify the instance [54]. As explained in KNN algorithm, we are using VIMA vectors of 8 KB and floating-point single precision, which gives us vectors with 2048 positions. Additionally, the number of neurons in the input layer is the number of features presented on the instances, while the hidden layer contains half of it. 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. If the hidden layer presents too few or too many neurons it may not identify properly the relevant features or it may consider every feature as being 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 6. In this work, we are considering that the NN is already trained, doing just the inference of the instances as the weights were trained and disregarding any other parameter of training or classification.

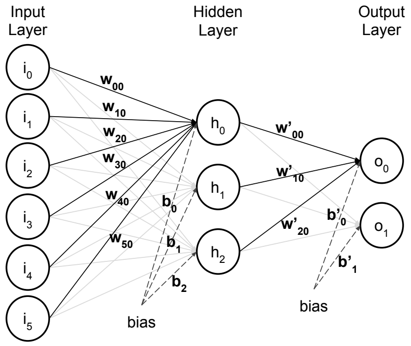


Fig. 6: Representation of an ANN.

If the instances are smaller than the VIMA vector, we compute a single instance at a time, as explained for the KNN algorithm. If the instances' size is equal to or greater than the VIMA vector, this transformation will not be necessary.

To obtain the hidden layer's activation values, first, we must use the Intrinsics-VIMA function *_vim2K_fmuls()* to multiply the input features and weight values (w_{xy}). Then, we use the function *_vim2K_fcums()* to accumulate the resultant values and store them in the vector of the hidden layer activation values. The algorithm repeats this operation for each neuron in the hidden layer. This hidden layer vector will store the activation values of all the instances sequentially. After calculating all the instances, we add the bias vector for all the neurons in the layer (the bias is a value to be added or subtracted to an activation value factor to adjust it and reduce errors) using the function *_vim2K_fadds()*. Finally, we use the function *_vim2K_fmaxs()* to apply the the activation function. In this work, we are considering Rectified Linear Unit (ReLU) as an activation function. Thus the hidden layer vector is operated with a zeroed vector, and every negative value is replaced by zero.

Similar to the input layer computation, we repeat the same steps for the hidden layer present in the MLP. Considering the varying number of weights for each layer, we must use specific masks to operate with each neuron separately, as depicted in Figure 7.

Since we consider only two types of labels, negative and positive, the output layer will have two neurons. Thus, two

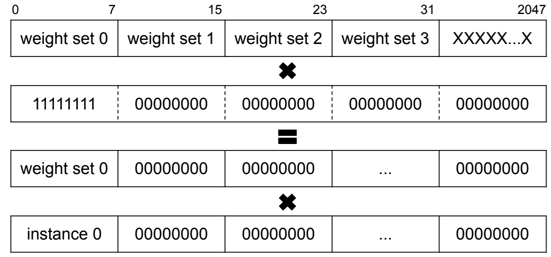


Fig. 7: Example of a VIMA vector with four sets of weights for instances representing 8 features.

sets of weights (w'_{xy}) are defined, both sets with the same size as the hidden layer and referring to the connections between the hidden and output layers. In the last, there are just two activation values and a Softmax activation function [55] must be applied to them to transform these values in probabilities. The higher probability corresponds to the label most likely to classify the instance. This final step does not use Intrinsics-VIMA functions.

VI. EXPERIMENTAL EVALUATION OF VIMA

This section presents the methodology and the simulation results for our ML kernel implementations.

A. *Methodology and Simulation Setup*

Computer architects often use simulators when evaluating new architectures. Compared to analytical models, simulators are more accurate, considering the high complexity of computer systems. Besides, simulators are faster and cheaper to implement new models if compared to prototyping. To evaluate our proposal, we adopted SiNUCA [56], a open-source cycle-accurate simulator. SiNUCA enables us to model our custom smart-memory architecture with FUs, a cache memory, and configurable operation size. Table II shows the main parameters used for our model.

TABLE II: 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;
	4 GB; DRAM@1666 MHz; 4-links@8 GHz; Inst. lat. 1 CPU cycle
	8 B burst width at 2.5:1 core-to-bus freq. ratio; Closed-row policy;
	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;
	256 int. units: alu, mul. and div. (8-12-28 cycles for 8 KB pipelined)
	256 fp. units: alu, mul. and div. (13-13-28 cycle for 8 KB pipelined);
	VIMA cache: 64 KB (8 lines), fully assoc., 2-cycle (1-tag, 1-per data);
	Dynamic energy: 194pJ per line access; Static power: 134mW;

x86 baseline: We inspired our baseline architecture in the Intel Sandy Bridge processor micro-architecture and referred to as x86. We modeled the ISA with AVX-512 instruction set capabilities besides all x86 ISA instructions. Furthermore, we use a 3D-stacked memory as the main memory.

VIMA architectures: To provide two scenarios for comparison, we propose using near-data operations over vectors of 8 KB. In this approach, we implemented the NEON ISA near-data. The x86 processor triggers these VIMA instructions. VIMA 8 KB mechanism and its 64 KB cache memory are estimated as 1.5W at 1 GHz with 32nm technological node.

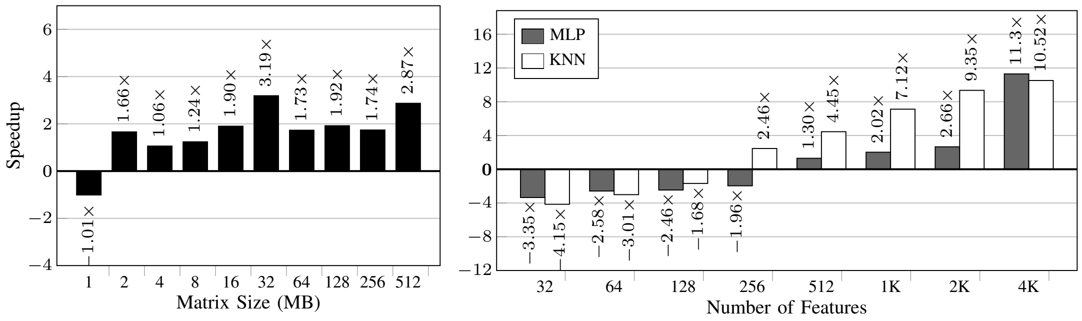
Benchmark: In our experiments, we evaluate KNN, MLP and convolution kernels. We used 4096 instances for MLP, 32768 training instances, 256 test instances, and 9 neighbors for KNN, varying the number of features for both applications (32, 64, 128, 256, 512, 1024, 2048, and 4096). For the convolution benchmark, we vary matrix dimensions (512×512, 724×724, 1024×1024, 1448×1448, 2048×2048, 2896×2896, 4096×4096, 5794×5794, 8192×8192, and 11648×11648). Our evaluations focus on architecture efficiency, not on the accuracy of each classification algorithm. Thus, the results will be shown in terms of speedup and energy savings.

In order to evaluate the energy consumption in our models, similar to other related work, we used CACTI and Multicore Power, Area, and Timing (McPAT) tools. Both were used to measure the cost of hardware on power, area, and timing parameters depending on their circuitry characteristics [57].

B. *Execution Time Results*

Figure 8(a) presents speedup results for the convolution algorithm described on Code 4 over matrices from size 512×512 to 11648×11648. The speedup for the convolution is not linear. It depends on the vector fill rate and the x86 baseline implementation time, which varies whenever the cache is more or less useful. We evaluated with the larger matrix of 11648×11648 that occupies 512 MB of memory, which still makes fair usage of the cache hierarchy of x86. Nevertheless, sizes greater than 16 MB slightly better utilizes the VIMA vectors, achieving thus the maximum performance.

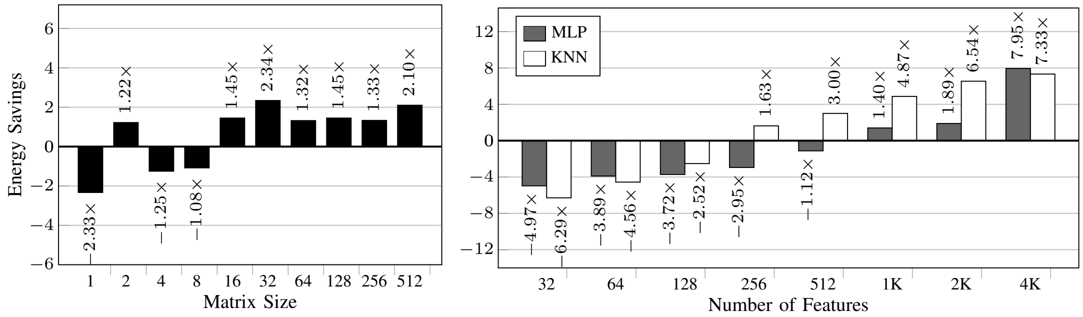
Figure 8(b) presents speedup results for MLP and KNN algorithms. Both algorithms start to present better results for VIMA when increasing memory usage. MLP and KNN exceed cache size with 512 and 256 features, respectively, using 32 MB of memory. When the memory footprint exceeds x86 cache memory size, the Advanced Vector Extensions (AVX) implementation starts to spend more time and energy in cache line replacements in comparison with VIMA. However, while it does not reaches this memory footprint, there is no speedup over the baseline, as we can observe for MLP with up to 256 features and for KNN with up to 128 features. Nevertheless, both algorithms have different behavior. Thus the speedup is more evident in KNN due to its quadratic complexity. On the other hand, MLP has linear complexity, achieving better results only when evaluating with a more significant amount of data, for example, with 4096 features (although using fewer features it presents a slow down up to 5× compared to the baseline).



(a) Convolution

(b) KNN and MLP

Fig. 8: Speedup results over baseline for (a) Convolution varying matrix size, (b) MLP and KNN varying number of features.



(a) Convolution

(b) KNN

Fig. 9: Energy savings of VIMA over baseline for (a) Convolution varying matrix size, (b) MLP and KNN varying number of features and neighbors.

C. Energy Results

Figure 9(a) presents the energy efficiency for the convolution, which follows the speedup pattern. The gains are higher when a matrix row fits perfectly into a VIMA vector, spending just half of the energy compared to the baseline.

For MLP and KNN algorithms, depicted in Figure 9(b), the energy savings are proportional to the speedup. It is possible to reduce in 7× the energy consumption using VIMA compared to the baseline. However, there are no energy savings for MLP and KNN with lower number of features, i.e. until 512 and 128, respectively. As we can observe in the graphic, VIMA can consume up to 6× more energy than AVX in these cases.

The energy savings achieved by VIMA depends directly on memory usage and algorithm behavior. Whenever the memory footprint fits inside the x86 cache memory, the processor presents higher efficiency. In contrast, 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.

VII. CONCLUSIONS AND FINAL CONSIDERATIONS

Considering the memory-wall problem, several approaches to NDP are emerging in the last years. Concurrently, ML algorithms are getting higher importance when analyzing large volumes of data. In this paper, we propose the migration of ML kernels to a vector execution near-data system to achieve high speedup with low energy consumption.

Using our Intrinsics-VIMA library extension, we could achieve a speedup of up to 10× for KNN, 11× for MLP, and 3× for convolution. Meanwhile, we obtained energy savings of 7× for KNN, $\sim$ 8× for MLP, and 2× for convolution compared to a baseline line system with x86.

Although we emphasize ML algorithms, other programs that rely on similar data access behavior shall benefit from VIMA. In general, it is expected a higher performance for algorithms that have streaming and coalescent data access behavior with low data reuse and a memory footprint bigger than the cache memory hierarchy capacity

As future work, we consider extending the migration to other ML algorithms, including its training phase and improving the Intrinsics-VIMA library to achieve better performance.

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].

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012.

[2] A. Rakotomamonjy, "Variable selection using svm-based criteria," *Journal of machine learning research*, vol. 3, no. Mar, 2003.

[3] M. W. Gardner and S. Dorling, "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences," *Atmospheric environment*, vol. 32, no. 14-15, 1998.

[4] L. E. Peterson, "K-nearest neighbor," *Scholarpedia*, vol. 4, no. 2, 2009.

[5] T. G. Dietterich, "Ensemble methods in machine learning," in *Int. workshop on multiple classifier systems*, 2000.

---

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

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

[6] A. Boroumand, S. Ghose *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[7] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, 1995.

[8] E. Nurvitadhi, J. Sim *et al.*, "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)*. IEEE, 2016, pp. 1–4.

[9] K. Kara, D. Alistarh *et al.*, "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)*. IEEE, 2017, pp. 160–167.

[10] M. Hashemi, E. Ebrahimi *et al.*, "Accelerating dependent cache misses with an enhanced memory controller," in *Int. Symp. on Computer Architecture (ISCA)*, 2016.

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

[12] M. K. Qureshi, A. Jaleel *et al.*, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, 2007.

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

[14] D. Patterson, T. Anderson *et al.*, "A case for intelligent ram," *IEEE micro*, vol. 17, no. 2, 1997.

[15] M. Gao, J. Pu *et al.*, "Tetris: Scalable and efficient neural network acceleration with 3d memory," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, 2017.

[16] J. Ahn, S. Hong *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, 2016.

[17] R. Nair, S. F. Antao *et al.*, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, 2015.

[18] M. A. Alves, M. Diener *et al.*, "Large vector extensions inside the hmc," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2016.

[19] P. C. Santos, G. F. Oliveira *et al.*, "Operand size reconfiguration for big data processing in memory," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2017.

[20] G. F. Oliveira, P. C. Santos *et al.*, "Nim: An hmc-based machine for neuron computation," in *Int. Symp. on Applied Reconfigurable Computing*, 2017.

[21] P. C. Santos, G. F. Oliveira *et al.*, "Processing in 3d memories to speed up operations on complex data structures," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*. IEEE, 2018.

[22] J. a. P. Lima, P. C. Santos *et al.*, "Design space exploration for pim architectures in 3d-stacked memories," in *Proceedings of the Computing Frontiers Conference*. ACM, 2018.

[23] A. S. Cordeiro, T. R. Kepe *et al.*, "Intrinsics-hmc: An automatic trace generator for simulations of processing-in-memory instructions," *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*, 2017.

[24] E. Azarkhish, D. Rossi *et al.*, "Neurostream: Scalable and energy efficient deep learning with smart memory cubes," *Trans. on Parallel & Distributed Systems*, 2018.

[25] F. Schuiki, M. Schaffner *et al.*, "A scalable near-memory architecture for training deep neural networks on large in-memory datasets," *arXiv preprint arXiv:1803.04783*, 2018.

[26] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Parallel Architecture and Compilation (PACT)*, 2015.

[27] M. Thottethodi, T. Vijaykumar *et al.*, "Millipede: Die-stacked memory optimizations for big data machine learning analytics," in *Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2018.

[28] J. Liu, H. Zhao *et al.*, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *Int. Symp. on Microarchitecture (MICRO)*, 2018.

[29] A. Ganguly, V. Singh *et al.*, "Memory-system requirements for convolutional neural networks," in *Proceedings of the Int. Symp. on Memory Systems*, 2018.

[30] C. Min, J. Mao *et al.*, "Neuralhmc: an efficient hmc-based accelerator for deep neural networks," in *Asia and South Pacific Design Automation Conf. (ASPDAC)*, 2019.

[31] S. Cadambi, A. Majumdar *et al.*, "A programmable parallel accelerator for learning and classification," in *Int. Conf. on Parallel architectures and Compilation Techniques (PACT)*, 2010.

[32] L. Xu, D. P. Zhang, and N. Jayasena, "Scaling deep learning on multiple in-memory processors," in *Workshop on Near-Data Processing*, 2015.

[33] J. P. C. de Lima, P. C. Santos *et al.*, "Exploiting reconfigurable vector processing for energy-efficient computation in 3d-stacked memories," in *Int. Symp. on Applied Reconfigurable Computing*, 2019.

[34] D. Gao, T. Shen, and C. Zhuo, "A design framework for processing-in-memory accelerator," in *Int. Workshop on System Level Interconnect Prediction (SLIP)*, 2018.

[35] Q. Deng, L. Jiang *et al.*, "Dracc: a dram based accelerator for accurate cnn inference," in *Design Automation Conf. (DAC)*, 2018.

[36] S. Li, D. Niu *et al.*, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Int. Symp. on Microarchitecture*, 2017.

[37] Q. Deng, Y. Zhang *et al.*, "Lacc: Exploiting lookup table-based fast and accurate vector multiplication in dram-based cnn accelerator," in *Design Automation Conf. (DAC)*, 2019.

[38] J. Sim, H. Seol, and L.-S. Kim, "Nid: processing binary convolutional neural network in commodity dram," in *Int. Conf. on Computer-Aided Design (ICCAD)*, 2018.

[39] C. Sudarshan, J. Lappas *et al.*, "An in-dram neural network processing engine," in *Int. Symp. on Circuits and Systems (ISCAS)*, 2019.

[40] D. G. Elliott, M. Stumm *et al.*, "Computational ram: Implementing processors in memory," *IEEE Design & Test of Computers*, vol. 16, 1999.

[41] J. V. Olmen, A. Mercha *et al.*, "3D stacked IC demonstration using a through silicon via first approach," in *Int. Electron Devices Meeting*, 2008.

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

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

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

[45] Hybrid Memory Cube Consortium, "Hybrid memory cube specification rev. 2.0," 2013, http://www.hybridmemorycube.org/.

[46] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Symp. on VLSI Technology*, 2012.

[47] J. Pawlowski, "Hybrid memory cube (hmc)," *Hot Chips*, vol. 23, 2011.

[48] C. Lomont, "Introduction to intel advanced vector extensions," *Intel White Paper*, 2011.

[49] I. Coorporation, "Intel 64 and ia-32 architectures optimization reference manual," 2009.

[50] B. McDanel, S. Teerapittayanon, and H. Kung, "Embedded binarized neural networks," *arXiv preprint arXiv:1709.02260*, 2017.

[51] J. Qiu, J. Wang *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35.

[52] Y. Tian, K. Pei *et al.*, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 303–314.

[53] S. Afonso, A. Acosta, and F. Almeida, "Automatic acceleration of stencil codes in android devices," in *Int. Conf. on Algorithms and Architectures for Parallel Processing*, 2017.

[54] T. M. Mitchell and M. Learning, "Mcgraw-hill science," *Engineering/Math*, 1997.

[55] C. M. Bishop *et al.*, *Neural networks for pattern recognition*. Oxford university press, 1995.

[56] M. A. Z. Alves, C. Villavieja *et al.*, "Sinuca: A validated micro-architecture simulator." in *HPCC/CSS/ICESS*, 2015, pp. 605–610.

[57] S. Li, J. H. Ahn *et al.*, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469–480.

SAIRO RAONÍ DOS SANTOS

NEAR-DATA BLOOM FILTERS FOR EFFICIENT DATA FETCH

(*versão pré-defesa, compilada em 8 de dezembro de 2020*)

Documento apresentado como requisito parcial ao exame de qualificação de Doutorado no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Marco Antônio Zanata Alves.

CURITIBA PR

2020

RESUMO

Constantes avanços na tecnologia de processadores significaram processamento mais rápido por décadas. Porém, as tecnologias de memória usadas pela maioria dos computadores não acompanharam tais avanços. Este desequilíbrio trouxe o problema conhecido como memory wall. A vazão relativamente baixa na transferência de dados entre memória e processador dificulta o uso pleno das capacidades de processamento do processador. O conceito de memórias inteligentes propõe inverter a prática de levar ao processador todos os dados que devem ser computados ao integrar elementos de processamento junto à memória. Assim, tarefas baseadas em processamento de dados podem ser realizadas com movimento reduzido, simultaneamente diminuindo o consumo de energia e melhorando o desempenho de tais aplicações. Memórias 3D são os mais populares dispositivos com capacidade de processamento em memória. Elas consistem em várias camadas de DRAM interconectadas verticalmente e divididas em canais administrados por controladores independentes. A base dessa pilha é uma camada lógica capaz de integrar dispositivos de processamento. Este tipo de design possibilita incluir novos elementos físicos no chip de memória, como registradores, unidades funcionais e memória adicional. A Vector-In-Memory Architecture (VIMA) é uma arquitetura que propõe incluir unidades funcionais vetoriais e uma pequena memória cache à camada lógica de uma memória 3D. Usando vetorização, a arquitetura utiliza boa parte do paralelismo da memória 3D e oferece, com a memória cache, a possibilidade de reusar linhas de dados vetorizados. Simulações indicam ganhos significativos em tempo de execução e consumo de energia ao executar kernels simples de álgebra e estrutura de dados, além de machine learning. Outra estratégia comum para mitigar problemas como o memory wall tem sido o uso de aceleradores implementados em hardware. Um candidato comum à implementação em hardware é o bloom filter, uma estrutura de dados usada por muitos tipos de aplicações para filtrar dados pertencentes a determinados conjuntos. Bloom filters são comumente usados por antivírus, algoritmos de sequenciamento de DNA e sistemas de gerenciamento de bancos de dados. Considerando os recursos da VIMA, este trabalho propõe investigar as possibilidades de implementação de bloom filters em hardware com o objetivo de agir como acelerador para diversos tipos de aplicações que se beneficiam deste tipo de estrutura. Esta pesquisa estabelece a hipótese de que, com uma implementação da estrutura em hardware capaz de explorar o paralelismo de acesso à memória considerado pela arquitetura, aplicações que fazem uso de bloom filters podem ter consumo de energia reduzido e execução mais rápida de tarefas altamente dependentes de dados. Palavras-chave: processamento próximo à memória.

bloom filter. big data.

ABSTRACT

Constant advancements in processor technology have translated into faster processing for decades. However, memory technology used by most computers has not kept up with such advances. This imbalance has caused the issue known as the memory wall. The relatively low throughput of data between memory and processor renders the processor unable to fully utilize its own resources. The processing-in-memory concept proposes inverting the practice of moving all data that must be processed to the processor by integrating processing elements in the memory device. Thus, tasks based on data processing can be performed with reduced data movement, simultaneously lowering energy consumption and improving the performance of such applications. 3D-stacked memories are the most common processing-in-memory-capable devices. They consist of several vertically connected DRAM chips logically divided in channels that are controlled independently by vault controllers. The lowest layer of the device houses processing elements that are able to perform simple arithmetic operations on data stored in the device. This type of device allows the inclusion of additional logical elements to the same structure, such as registers, functional units and additional memory. Vector-In-Memory Architecture (VIMA) is an architecture that proposes to include vector units and a small cache to the device. By using data vectorization, it leverages the intrisic parallelism of the device and, with its cache memory, offers the possibility of data reuse. Simulation results show significant improvements in execution time and energy consumption while running several simple operations and machine learning algorithms. Another common strategy to mitigate issues like the memory wall has been the use of accelerators implemented on hardware, aimed at repetitive tasks in specific applications. These elements work as co-processors or throughput amplifiers. A common candidate to hardware implementation is the bloom filter, a data structure used by several applications to filter data in certain sets. Bloom filters are commonly used by antiviruses, genome sequencing algorithms, and database management systems. Considering the resources available to VIMA, this work proposes investigating the possibilities of implementing bloom filter functionality in VIMA with the goal of acting as an accelerator to several types of applications that benefit from this kind of structure. This research establishes the hypothesis that, with the support of such an implementation that is capable to leverage the capabilities of such an architecture, applications that use bloom filters may achieve lower energy consumption and faster execution of tasks that are highly dependent of data.

Keywords: near data processing. bloom filter. big data.

LISTA DE FIGURAS

1.1 Block diagram of a 3D-stacked memory 9

2.1 The von Neumann architecture 12
2.2 Memory scaling trends. 14
2.3 Insertion of elements into an empty Bloom filter. 18
2.4 Checking element membership 19

3.1 3D-stacked memory module with the VIMA architecture 22
3.2 Speedup of VIMA normalized to baseline 25
3.3 Speedup and energy of VIMA and AVX multithread normalized to baseline . . . 26

5.1 RGB data deinterleaving with the VLD3 instruction 36
5.2 Initial state of vectorized data for the shift-add-xor hash function 37
5.3 Initial state of vectorized data for the shift-add-xor hash function 37

LISTA DE TABELAS

2.1 Evolution of SIMD instruction sets. Adapted from Al Hasib (2018). 15
2.2 Bloom filter stats considering a 16 KB bit array (Lyons and Brooks, 2009).. . . . 20

3.1 Baseline and VIMA system configuration. 24

4.1 Table of papers. 32

5.1 Planning chart. 40

LISTA DE ACRÔNIMOS

ASIC	Application-Specific Integrated Circuit
AVX	Advanced Vector Extensions
BLAST	Basic Local Alignment Search Tool
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DDR	Double Data Rate
DRAM	Dynamic Random Access Memory
DU	Data Deduplication
FP	Floating-point
FPGA	Field Programmable Gate Array
FU	Functional Unit
GPU	Graphic Processing Unit
HBM	High Bandwidth Memory
HIVE	HMC Instruction Vector Extensions
HMC	Hybrid Memory Cube
HPC	High Performance Computing
ISA	Instruction Set Architecture
IoT	Internet of Things
JEDEC	Joint Electron Device Engineering Council
LRU	Least Recently Used
ML	Machine Learning
MOB	Memory Order Buffer
NDP	Near-Data Processing
NIDS	Network Intrusion Detection Systems
NN	Neural Network
OrCS	Ordinary Computer Simulator
PIM	Processing-In-Memory
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

SUMÁRIO

1 INTRODUCTION 8
2 BACKGROUND 12
2.1 THE VON NEUMANN ARCHITECTURE 12
2.2 MEMORY WALL 13
2.3 SINGLE INSTRUCTION MULTIPLE DATA INSTRUCTIONS 15
2.4 NEAR-DATA PROCESSING. 16
2.5 BLOOM FILTERS 17
3 VIMA: VECTOR PROCESSING AND DATA REUSE INSIDE THE MEMORY 21
3.1 OVERVIEW. 21
3.2 EXPERIMENTAL EVALUATION OF VIMA 23
4 RELATED WORK 27
4.1 NEAR-DATA PROCESSING. 27
4.2 BLOOM FILTERS 28
4.2.1 String Matching Applications. 28
4.2.2 Hardware Implementations 29
5 PROPOSAL 33
5.1 HYPOTHESIS AND OBJECTIVES 33
5.2 INITIAL DESIGN 34
5.3 RESEARCH PLAN. 38
5.4 PROPOSED SCHEDULE 39
5.5 METHODOLOGY 41
5.5.1 Ordinary Computer Simulator (OrCS) 41
5.5.2 Binary Generation. 41
REFERÊNCIAS 43

1 INTRODUCTION

Constant advances in processor technology have translated directly into faster processing for several decades. However, the technology used for most computer systems' main memory has not keep up with such advances (Chang, 2017). In the age of Big Data, as applications move toward a more data-centric behavior, as opposed to a computation-centric one, the issue of the performance gap between processor and memory worsens. The relatively low data transfer throughput between memory and processor makes it difficult for the processor to fully utilize its processing capabilities. Thus, emerges the problem widely known as the memory wall (Wulf and McKee, 1995).

The von Neumann architecture is the basic concept of all modern computers (von Neumann, 1945). It requires that any data necessary for computation be moved from the memory and placed within the processor before it can be processed. To avoid moving massive amounts of data from memory to the processor, researchers started proposing placing processing elements as close as possible to the memory (Balasubramonian et al., 2014). This approach has two highly desirable results: i) Reduced energy consumption by reducing the amount and distance of data movement, which accounts for up to 62.7% of total energy expenditure of a system (Boroumand et al., 2018); ii) Faster execution of data-centric applications, as the processor can offload a large chunk of its operations for this additional processing element to perform in parallel.

Researchers have proposed two main approaches to this: Near-Data Processing (NDP) and Processing-In-Memory (PIM). NDP attaches an additional processing logic device to the same interconnection structure as the memory chip. Meanwhile, PIM applies changes to the circuitry of memory chips and memory cells to perform operations over the stored data, avoiding the need for additional elements and taking advantage of the existing internal bandwidth of memory cells (Boroumand et al., 2018). Both approaches avoid off-chip data transfers.

3D-stacked memories are a novel main memory design that stacks up to eight layers of Dynamic Random Access Memories (DRAMs) on top of a layer that features processing capabilities (Hybrid Memory Cube Consortium, 2014; Hrusca, 2015). Due to the 3D layout, as depicted in Figure 1.1, these memory chips offer high parallelism and low-latency access to the stored data.

Some 3D-stacked memories are NDP-capable due to the inclusion of data processing elements to their logic layer. Moreover, these devices enable researchers to explore new possibilities for NDP, as they allow for the inclusion of logical elements near-data, such as registers, Functional Units (FUs) or accelerators. Therefore, a NDP design can still follow the von Neumann model by placing entire processors near the data. This approach, however, may increase complexity. Another possible approach is to extend the model by placing FUs near-data, which avoids some of these issues and allows processors to continue handling tasks they excel at,

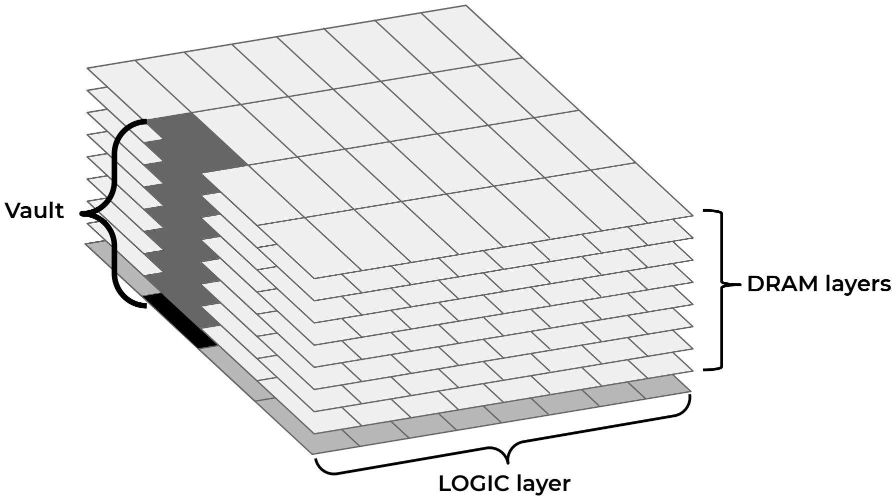


Figura 1.1: Block diagram of a 3D-stacked memory.

such as fetching and decoding complex instructions, predicting the outcome of branches, among other functions.

Vector-In-Memory Architecture (VIMA) is an architecture that proposes adding vector units and a small cache memory in the logic layer of 3D-stacked memory chips. This architecture uses large data vectors to benefit from the data access parallelism that is intrinsic to the 3D configuration of the memory chip and offers, with the cache memory, the possibility of short term data reuse. Simulation results indicate significant improvements in execution time and energy consumption when executing simple benchmarks and selected machine learning algorithms.

Another common strategy to mitigate issues like the memory-wall and dark-silicon has been hardware accelerators for specific applications (Fang et al., 2020). Such elements work as co-processors controlled by the host processor or transparently as throughput amplifiers. They are used to assist applications that require massive and constant data movement, with stream behavior whose memory footprint exceeds the size of the cache hierarchy (such as machine learning algorithms, anti-viruses and genome sequencing programs).

Considering the applications with data stream behavior, Bloom filter algorithms are very useful to filter data in a simple and effective way. Bloom filters are a probabilistic data structure often used by applications to check data for membership of a specific set. They use an array bit and a set of hash functions that can be adjusted according to the size of the data set being represented to control for false-positive rates. By using Bloom filters for such tasks, storage costs may be reduced by up to 70% in comparison to other approaches to membership checking with constant time complexity. They are often used in real-time applications and applications that deal with huge data sets, such as network intrusion detection systems and database management systems.

This work aims to investigate the possibilities of implementing Bloom filters as hardware as part of an NDP architecture with a 3D-stacked memory. We believe such an implementation can be used as an accelerator to aid applications in several domains that benefit from this type of data structure. We hypothesize that, with a hardware implementation that can exploit the data access parallelism available in this type of device, applications that use Bloom filters can achieve reduced energy consumption and faster execution of highly data-centric tasks.

Thus, our objective is to provide a solution that can be used by applications that often must parse through vast data sets and benefit from an efficient hardware implementation to aid this task. By providing such an accelerator, as the task relies on this new element for processing, rather than on a processor core, we wish to mitigate performance limitations when processing large amounts of data caused by issues such as the von Neumann bottleneck and the memory wall.

This approach is aimed specifically at supporting applications dealing with large volumes of data. While NDP architectures like VIMA can be very effective at mitigating issues related to data movement, other ubiquitous strategies like cache memories are already efficient whenever the memory footprint of an application is small enough that it fits inside the cache hierarchy (Alves et al., 2016). However, as the largest level of the cache hierarchy becomes overwhelmed, it ceases to provide any benefit to data access latency times and, from that point onwards, an NDP architecture can possibly achieve both faster processing and reduced energy consumption.

In considering NDP architectures that propose adding elements to 3D-stacked memory chips, one could argue that adding a full core to the architecture could be preferable to a simpler setup such as the one VIMA proposes. However, that would cause a number of issues. A full core would add a level of complexity to the design that would outweigh the added functionality, as it would introduce several elements that would possibly need to be consistently coherent with all the cores in the host processor. Thus, simpler architectures are preferable in that they are able to achieve a significant processing speed-up while maintaining a low level of complexity.

We expect to face a number of challenges in pursuing research in this direction. While the Bloom filter is a relatively simple data structure, providing an effective hardware implementation generic enough for multiple applications in the fashion we propose must consider several requirements. The following is a non-exhaustive list of some of the challenges we foresee:

- **Data alignment**: We believe that it is essential to consider how application data is organized in the memory for optimized processing of sets with huge item counts. If the processing of items for both creation and consulting of Bloom filters is not efficient, throughput might not be high enough to justify the usage of such a solution.

- **Application migration**: We do not expect to support every computer application. However, we plan to address applications that face the memory wall, including most big-data applications from several different domains.

- **Validation**: We must rely on simulators for test and validation of any designs we propose, like most research in computer organization and architecture.

- **Design choices**: We foresee multiple possible designs, from ASIC to VIMA extension. Nevertheless, it is essential that our resulting proposal carefully considers all design choices regarding Bloom filters, such as supported data set size, number of hash functions, and desired false-positive rates.

- **Security**: We must provide some protection and isolation of the Bloom filters from malicious activities, especially as we wish to support applications like anti-viruses and network intrusion detection systems.

- **Scope**: We expect to have to narrow our application domain scope to those that are feasible considering architectural and timeframe constraints. Thus, it may not be possible to support every kind of Big Data application that can benefit from Bloom filters.

The remainder of this document is organized as follows. Chapter 2 discusses many concepts related to the subject of this research proposal. Namely, the von Neumann architecture, the memory wall, Single Instruction Multiple Data (SIMD) instructions and bloom filters. Chapter 3 gives an overview of VIMA, its main components, behavior and simulation results. Chapter 4 goes over some of the existing work related to PIM and NDP architectures and how they relate and compare to VIMA and also discusses research related to bloom filters, their most common applications and existing hardware implementations. Chapter 5 describes our proposal, establishes research hypotheses and objectives, and lays out a tentative schedule for the research.

2 BACKGROUND

This chapter presents our proposal's main concepts, such as the von Neumann architecture, the memory wall, Single Instruction Multiple Data (SIMD) instructions, Near-Data Processing (NDP), and Bloom filters.

2.1 THE VON NEUMANN ARCHITECTURE

The von Neumann architecture (von Neumann, 1945) describes a computer that consists of four fundamental components: a processing component that performs logic and arithmetic operations; a control component that manages a stream of instructions; a storage component that houses data the control component instructs the processing component to handle; and an input/output component.

It defines that tasks for a computer to complete are described as lists of individual instructions performed by its processing component. This list is managed by its control component, informing the processing component what instruction to perform next. These two items are often housed in the processor known as Central Processing Unit (CPU).

Each instruction includes, aside from the operation the processing component must perform, what data it must use or affect. The control component must also take note of this, as all data necessary for an instruction to be performed must be available for the processing component at the time of its execution. This data must be fetched from the storage component, also known as memory, and the operation result may be stored in it. Once a processing task is completed, its outcome may be presented to a user through the input/output component. Any data produced or used by the input or output components must also be placed in the memory.

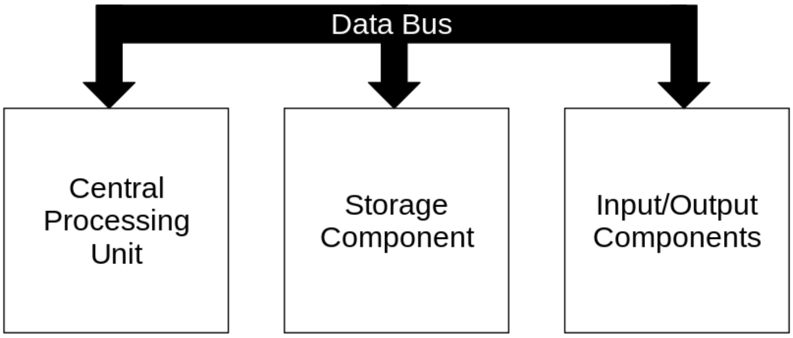


Figura 2.1: The von Neumann architecture.

Figure 2.1 shows a simplified representation of the described architecture and highlights an issue known as the von Neumann bottleneck. Since all components are connected by a bus through which all data movement must take place, the speed at which the overall system can function is limited by the speed of the bus itself. As more extensive and complex tasks started being handled by computers, program instructions and data stored in the memory became a problem. The processor relies on the bus that connected it to the memory to fetch each instruction and data. This distance between logic and storage, together with the high latency on memory systems, was the initial motivation for placing instruction caches inside the processor, as the instruction and data fetching became a bottleneck.

Cache memories are small Static Random Access Memory (SRAM) chips that hold several times less data than the main memory, but that offer much quicker access to it. These are present in practically every modern computer processor designed in the last few decades. These cache memories are placed inside the processor. Thus, the instructions they store can be accessed much faster because of the faster storage technology and because of this placement in the architecture.

2.2 MEMORY WALL

Modern computers' processing capabilities improved sharply, as predicted by Moore's Law (Schaller, 1997). However, the memory technology was unable to evolve at the same pace, lagging behind significantly. As the processor needs to wait for data fetch before operating, this caused a bottleneck that is commonly called memory wall (Wulf and McKee, 1995).

The most common type of memory used in modern computers is the Dynamic Random Access Memory (DRAM), a technology that offers cheap storage requiring only one transistor and one capacitor per bit. On the other hand, it suffers from high access latency, meaning data takes multiple steps and a long time to travel from the memory cell to the processor (Jacob et al., 2010)

There are several reasons why this communication latency is so high. For instance, the DRAM employs capacitor cells to store information, which are charge-based and thus must be refreshed regularly. Nevertheless, such voltage decay requires sense amplifiers to retrieve data. Furthermore, its access protocol consists of several phases that require waiting for specific clock counts. Lastly, it is placed off of the processor chip, which means data must travel through the data bus to get to the processor.

Figure 2.2, adapted from Chang et al. Chang (2017), illustrates DRAM technology scaling trends over two decades. The main strategy to mitigate the issue has been to increase parallelism inside the DRAM, with the addition of multiple channels, ranks, and banks that can be accessed independently. Thus, while capacity and bandwidth improved by 128× and 20× respectively, latency only improved by 1.3×.

On the other hand, one avenue explored to reduce the latency caused by data movement was to keep data close to the processor through the use of data caches. These cache memories store data whenever the processor fetches it from the main memory. If a cache memory is full, its replacement policy chooses a block to evict, freeing space for the incoming data. The cache aims

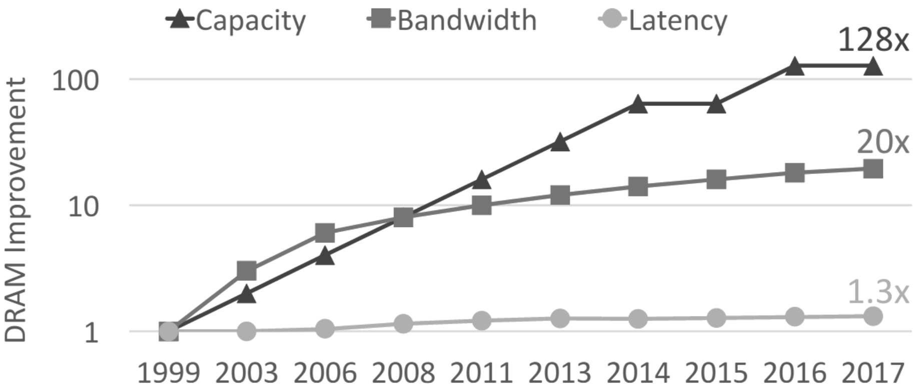


Figura 2.2: Memory scaling trends. Adapted from Chang (2017).

to store data most likely to be sooner referenced again by the processor. This assessment is based on the concept of locality of reference (Jacob et al., 2010).

The locality of reference alludes to the fact that most applications' data usage patterns are not random but rather follow a simple behavior: they tend to access the same data more than once in a short time. They may also access nearby data addresses in the memory space. As long as applications follow this pattern, cache memories are effective at mitigating the memory wall issue. After some data is fetched from the main memory once, it is placed in the cache memory, and subsequent references to this data cause the processor to fetch it from the cache, thereby causing the access to have a small latency (Jacob et al., 2010).

However, this is not true for every application. Programs with other data access behavior patterns may not benefit from cache memories and experience the DRAM latency for most data accesses (Wulf and McKee, 1995; Balasubramonian et al., 2014; Hashemi et al., 2016). While data prefetching techniques may help mitigate this problem, they can also cause cache pollution if they mispredict the application behavior. At the same time, prefetchers still causes massive data movement through the memory hierarchy (Ebrahimi et al., 2009).

As applications become data-centric, their data-set tends not to fit inside the cache memories, making such memories useless for data-reuse and performance purposes. Machine learning algorithms, for instance, are notorious for their data-hungry behavior and may thus be primarily limited by the memory wall. This behavior is also found on other typical consumer applications, such as those that use 4K video streaming, virtual reality, 3D graphics, etc (Boroumand et al., 2018).

With this trend, another issue caused by data movement becomes apparent: the increased amount of energy it causes systems to consume. Moving data through the system consumes more energy than computation. Thus, as applications become memory-bound, the energy spent on moving this data back and forth becomes a problem (Balasubramonian et al., 2014). On average, about 62% of the total energy is spent when moving data from the memory to the processor (Boroumand et al., 2018).

Some efforts to reduce the impact of this issue have focused on making the processor more tolerant of latency by basically designing it to be capable on focusing on other necessary tasks as it waits for data to be transferred from the memory. One example is the non-blocking cache memory, which allows caches to fulfill other requests at the same time as it fetches data from the memory to respond to a cache miss. Multithreading is another strategy, which moves the control flow of the processor to a different thread in the event of a cache miss, while prefetching is a technique that tries to learn the pattern of memory requests issues by an application and uses that information to issue requests for data before it is actually needed. While these help the system tolerate some latency, they also generate more memory requests, which contributes to overwhelming the memory bandwidth and reintroduces a bottleneck (Efnusheva et al., 2017).

2.3 SINGLE INSTRUCTION MULTIPLE DATA INSTRUCTIONS

Single Instruction Multiple Data (SIMD) instructions enable systems to apply a specific transformation to several data points at once, as opposed to operating over (scalars) only one data point per instruction, thus improving the ratio of processed data per number of instructions. Supercomputers used this concept since the 1970s, and SIMD instruction sets started to appear on desktop computers in the 1990s, as home users became interested in machines powerful enough to handle audio and video processing and thus manufacturers began looking into ways to accelerate video processing (Lee, 1995).

The first widely known SIMD instruction set was MMX, which was included on Pentium processors in 1997 and used 64-bit registers. SIMD instruction sets are included in virtually every processor and have since become much more useful with bigger vectors and a wider variety of operations. Table 2.1, adapted from Al Hasib (2018), shows the evolution of SIMD instruction sets on Intel processors.

Tabela 2.1: Evolution of SIMD instruction sets. Adapted from Al Hasib (2018).

Year	**SIMD Instruction Set**	**Introduced In**	**Vector Size**
1997	MMX	Pentium MMX	64-bit
1999	SSE	Pentium III	128-bit
2000	SSE2	Pentium 4 Willamette	128-bit
2004	SSE3	Pentium 4 Prescott	128-bit
2007	SSE4.1	Penryn	128-bit
2009	SSE4.2	Nehalem	128-bit
2011	AVX	Sandy Bridge	256-bit
2013	AVX2	Haswell	256-bit
2016	AVX512	Knights Landing	512-bit

SIMD instruction sets provide improved performance, better utilization of resources, and improved energy efficiency (Al Hasib, 2018). This strategy assumes several constraints: i) It requires adding registers with larger storage capacity, commonly regarded as vector units, to the processor. ii) As implied by the term ’vector unit’, the data points are treated as a vector and stored in consecutive memory addresses. iii) The memory wall remains a limiting factor because this approach does not attempt to mitigate data movement from the memory to the processor. iv) Compilers can seldom automatically identify opportunities for transformation into SIMD instructions and thus require programmers explicitly to consider vector opportunities when coding.

Processor manufacturers provide ways for programmers to write vector enabled code. The intrinsics libraries offer routines that can be used on regular code and explicitly signal to the compiler that an operation must be performed with a vector unit. These routines calls are automatically translated into vector operations in the assembly code generated during compilation (Lomont, 2011).

SIMD instructions may achieve two desirable effects: i) An increase in processed data per instruction ratio, causing fewer accesses to the memory; ii) Improved usage of the parallelism capabilities of the memory device, reducing the overall memory latency.

2.4 NEAR-DATA PROCESSING

A different approach to dealing with the memory wall has been altering the traditional processor-memory architecture. Since the problem stems from the need to move data to the processor whenever an instruction requires it for execution, researchers started to investigate the possibility of inverting this relationship and performing the required transformation where the data is stored.

This concept emerged as an alternative to classical architectures, aiming to address issues related to computation and memory access limitations, such as the memory wall (Wulf and McKee, 1995) and dark silicon (Esmaeilzadeh et al., 2011). In this new approach, at least one processing element is included in the same chip as the memory and used to perform operations over the data, thus mitigating data movement between memory and processor. Such smart memories improve performance and reduce energy consumption simultaneously as they offer high parallelism and ensure low average latency during computation of tasks that apply high pressure to the memory (Patterson et al., 1997; Elliott et al., 1999).

These initiatives usually employ one of two main strategies: i) Applying changes to the circuitry of memory chips to enable them to perform operations on the stored data (Mutlu et al., 2019), thereby avoiding additional logic elements and taking advantage of the existing internal bandwidth of memory cells, known as Processing-In-Memory (PIM); ii) Offloading tasks to an additional processing logic device placed on the same interconnection structure as a 3D-stacked memory chip, thus exploiting the low-latency and high-bandwidth access capabilities inherent to this type of design, known as NDP. In this thesis proposal, our primary focus is on the NDP approach as it relies mainly on architectural modification and less on electrical grid changes (as opposed to the PIM approach).

Generally speaking, a 3D-stacked memory chip is composed of multiple stacked layers of DRAMs (usually 8) and a base layer that provides a logic layer, where a processing element can be integrated to operate elements inside the memory, as depicted in Figure 1.1. The memory is logically partitioned into independent vaults (usually 32 partitions), each further divided into independent DRAM banks (ranging from 8 to 16 banks per vault) distributed among DRAM layers and linked through Through-Silicon Vias (TSVs) (Olmen et al., 2008). The most well-known examples of 3D-stacked memories are Hybrid Memory Cube (HMC) (Hybrid Memory Cube Consortium, 2014), which has been discontinued, and High Bandwidth Memory (HBM) (Jun et al., 2017), the current commercial version in accordance to the Joint Electron Device Engineering Council (JEDEC).

The logic layer from the 3D-memory can implement processing elements to execute some operations to avoid some data movement to the processor. For HMC, the supported operations are atomic and consist of fetching data from a specific location, applying some modification to it, and storing the resulting data back in the same location (Nai and Kim, 2015). Since the device is logically partitioned in independent vaults, each managed by its vault controller, each processing element can only modify data stored in its vault. Such organization limits the utilization of these processing capabilities for more complex tasks and guarantees low latency and high bandwidth during data accesses (Mutlu et al., 2019; Hybrid Memory Cube Consortium, 2014; Jeddeloh and Keeth, 2012; Pawlowski, 2011).

The high bandwidth provided by the internal parallelism (Hybrid Memory Cube Consortium, 2013; Jeddeloh and Keeth, 2012; Pawlowski, 2011) achieved by its 3D integration technology along with the 32 vaults makes such architectures ideal for streaming and parallel applications, graphics processing, High Performance Computing (HPC), and networking. Generally, any application with coalescent memory accesses can benefit from it. Compared to the energy costs of Double Data Rate (DDR) memory technology, these devices require the same voltage level on average. However, 3D memories can achieve higher memory bandwidth, reaching up to 320 GB/s (Transcend, 2014; AMD, 2015), which makes them more energy-efficient (Hrusca, 2015).

NDP usage presents challenges, such as efficiently maintaining cache coherence and translating virtual addresses in the memory (Ghose et al., 2018), but 3D-stacked memory devices offer several exciting possibilities for new architectures. Such memories may include additional architectural elements such as sequencers, Functional Units (FUs), or even full processing cores, thereby giving these elements direct memory device access. Moreover, connecting such proposals next to the internal interconnection enables ideas that leverage accessing data within all vaults, further exploiting the parallelism and consequent bandwidth capabilities inherent to the device.

2.5 BLOOM FILTERS

Bloom filter algorithms aim to indicate whether an element is present in a set in a fast and memory-efficient way by using hash structures to represent the set of elements.

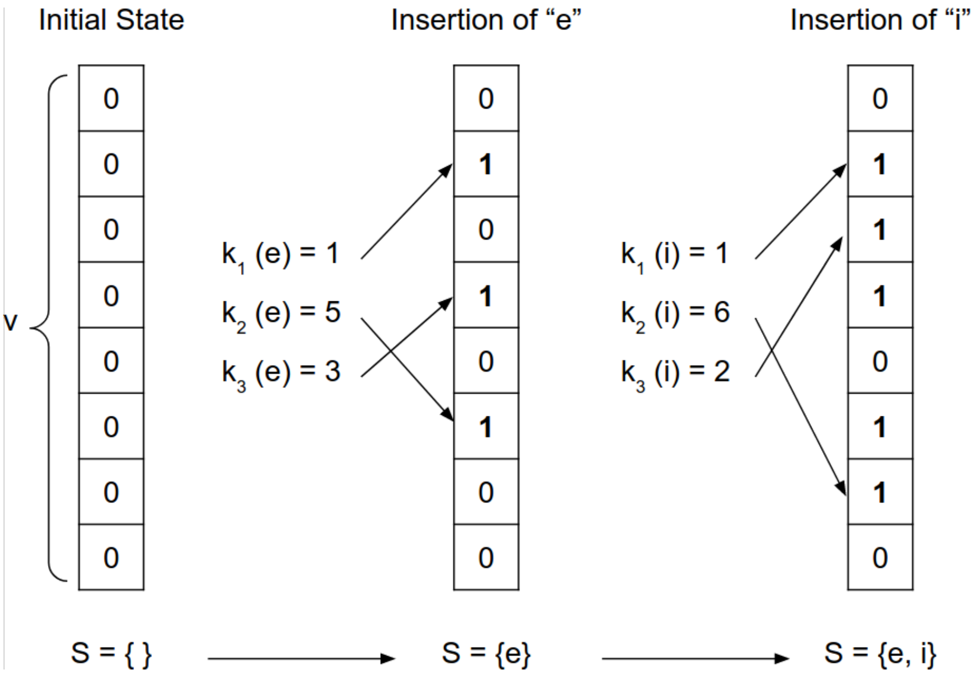


Figura 2.3: Insertion of elements into an empty Bloom filter. We consider $m = 8$ and $k = 3$.

A Bloom-filter consists of a bit array v of length m and a set of k_i independent hash functions. Consider one wishes to use a Bloom filter to represent a set of items S. Initially, all bits in v are set to zero, which signifies no items have been added to the structure yet. The insertion phase consists of iterating over every element $e \in S$ and applying the same operation over each one. The insertion operation entails applying all k_i hash functions to e and setting the bit at the position of v that corresponds to each hash result. Figure 2.3 shows an example considering a Bloom filter representing an empty set and the subsequent addition of two elements to the set.

The inquiry phase consists of a very similar process. Suppose we wish to check whether an element $n \in S$. To find out, we apply all k_i hash functions to n and check the bits at the positions corresponding to each result. Due to the mathematical properties of hash functions, if *any* of those bits are set to zero, we know for sure that $n \notin S$. On the other hand, if all bits are set to one, we cannot ensure that $n \in S$. False positives are more likely to occur due to hash collisions as more keys are added, and more bits are set equal to one. See Figure 2.4 for an example of each situation (Zengin and Schmidt, 2016). In general, false-positives may occur while false-negatives will never happen on Bloom filters.

The four variables that must be considered when designing a Bloom filter are the bit array length m, number of items n, number of i independent hash functions k_i and false positive probability p. Equations 2.1 and 2.2 show how all four variables influence each other (Santos et al., 2020).

$$m = -(\frac{n \times ln(p)}{(ln(2.0)^2)}) \tag{2.1}$$

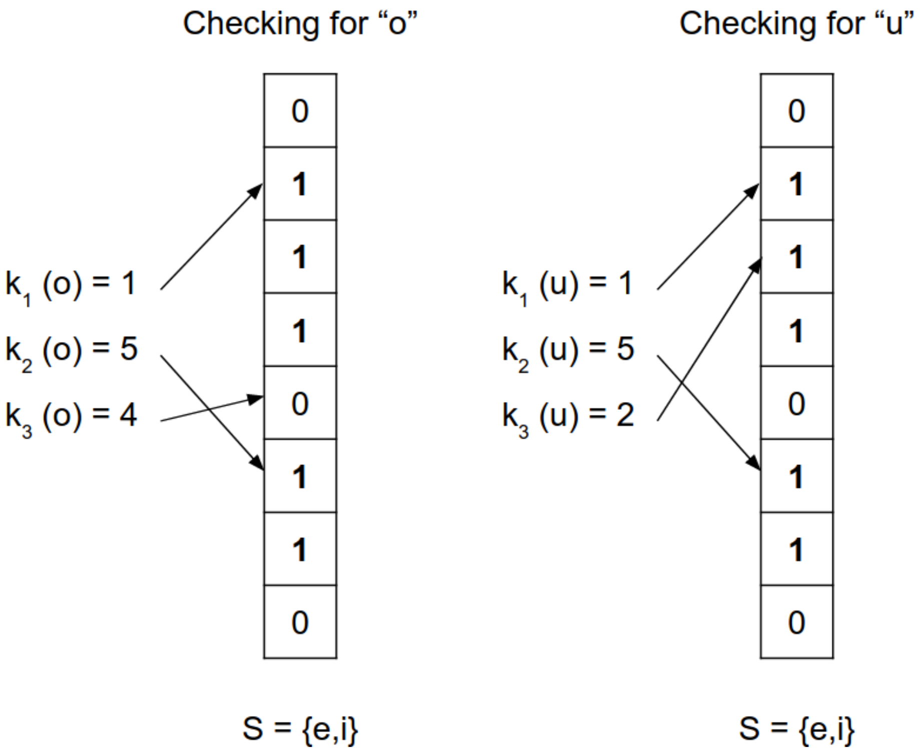


Figura 2.4: Checking element membership: "o" causes a negative result and "u" causes a false positive.

$$k = ln(2.0) \times \frac{m}{n} \tag{2.2}$$

The false-positive probability is one of the main aspects one must consider when designing a Bloom filter. Considering a hash function with perfect spread, the false positive probability p of a Bloom filter can be calculated with Equation 2.3 (Gupta and Batra, 2017).

$$p = (1 - e\frac{-kn}{m})^k \tag{2.3}$$

The false-positive rate of a Bloom filter is most affected by the number of hash functions, as the collision rate of each additional function causes false-positives to be less likely. However, as more bits are set per item, the structure becomes saturated with fewer items. According to (Lyons and Brooks, 2009), a Bloom filter is considered saturated or full when half of its bits have been set, as false positives become dramatically more likely if any more bits are set to 1. At this point it is also the as space efficient as possible, considering a fixed bit array length. Thus, if one wishes to lower the false positive rate of a Bloom filter considering a specific space requirement, this can be achieved by increasing the amount of hash functions it uses and decreasing the number of items it considers. Table 2.2, adapted from (Lyons and Brooks, 2009), considers a Bloom filter with a 16 KB bit array (m) and varies the number of hash functions k_i. Capacity n of the Bloom filter and false positive probability p vary accordingly.

Hash Functions (k)	Item Capacity (n)	False Positive Rate (p)
7	13500	< 1%
10	9000	< 0.1%
14	6500	< 0.01%

Tabela 2.2: Bloom filter stats considering a 16 KB bit array (Lyons and Brooks, 2009).

As the Bloom filter became a popular data structure in various application domains, several variants have been proposed, each addressing specific application needs and characteristics. According to (Patgiri et al., 2019), these are the more common Bloom filter variants:

- *Counting Bloom Filter*: instead of a single bit per index, the counting Bloom filter includes a counter of how many times an index has been set. This design modification permits the removal of elements from the set. When an inquiry happens, the structure returns the smallest count found out of all indices (Mcvicar et al., 2017).

- *Fingerprint Bloom Filter*: consider that the false positive rate of a counting Bloom filter is higher than that of a standard Bloom filter. The fingerprint Bloom filter stores small hashes instead of a counter, which reduces the probability of a false positive. Naturally, storage costs are higher than those of a standard or counting Bloom filter (Fan et al., 2014).

- *Hierarchical Bloom Filter*: the hierarchical Bloom filter employs a tree of separate Bloom filters and achieves a lower false-positive rate. It addresses the scalability issue of standard Bloom filters if the number of items in the set is not known at the time of creation, as it can expand as needed (Lu et al., 2011).

- *Multidimensional Bloom Filter*: stores bits on a structure with several dimensions instead of an array to reduce false-positive rate.

- *Compressed Bloom Filter*: a Bloom filter variant that is even more space-efficient than the standard Bloom filter, but with higher false-positive probability (Lyons and Brooks, 2009).

Observing that Bloom filters are relatively simple data structures suggests that such implementations can be useful as accelerators and co-processors to mitigate issues such as the von Neumann bottleneck and the memory wall. Their widespread use in applications dealing with large volumes of data will be further discussed in Section 4.2.

3 VIMA: VECTOR PROCESSING AND DATA REUSE INSIDE THE MEMORY

In this chapter we discuss Vector-In-Memory Architecture (VIMA), a novel general-purpose NDP architecture. VIMA interests us for several reasons: i) Its design features many components we find desirable for our research proposal. ii) While it is a general-purpose architecture, it keeps complexity low by avoiding the addition of a full processing core to the memory chip. iii) A complete environment for simulation of the VIMA architecture is available. Thus, we have selected it as a starting point for our research, upon which we hope to improve and from which we believe we will branch out as we progress.

The next sections give an overview of the architecture through a thorough explanation of its components and inner workings and discuss several benchmarks' simulation results compared to an x86 baseline.

3.1 OVERVIEW

VIMA adds general-purpose vector operation capabilities to 3D-memories to explore the data access parallelism inherent to this architecture, including a cache memory that enables fast reuse of vectorized data within the memory. Similar to other NDP approaches, VIMA obtains data from several independent memory vaults in parallel (Alves et al., 2016; Santos et al., 2017; Tomé et al., 2018). The main physical addition of VIMA compared to related work 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. Figure 3.1 shows VIMA inside a 3D-stacked memory.

Like in other technologies that employ vectorized data, such as Streaming SIMD Extensions (SSE), Advanced Vector Extensions (AVX) and NEON, VIMA instructions are inserted into the application by the compiler. During execution, VIMA instructions traverse the processor pipeline up to the execution stage like a regular memory instruction. They are then sent for execution in the 3D-stacked chip, avoiding data movement between memory and processor. VIMA instructions have a 3-operand format and operate over data vectors of 8 KB or 256 B. An 8 KB vector enables usage of the full parallelism of a 3D-stacked memory chip with 32 vaults and at least eight banks per vault, while a 256 B uses a single vault. The small cache memory of 64 KB inside VIMA stores up to eight 8 KB vectors or 256 256 B vectors. Our flexible design also allows the usage of smaller or larger data vectors, which would reduce or increase the parallelism inside the memory, respectively.

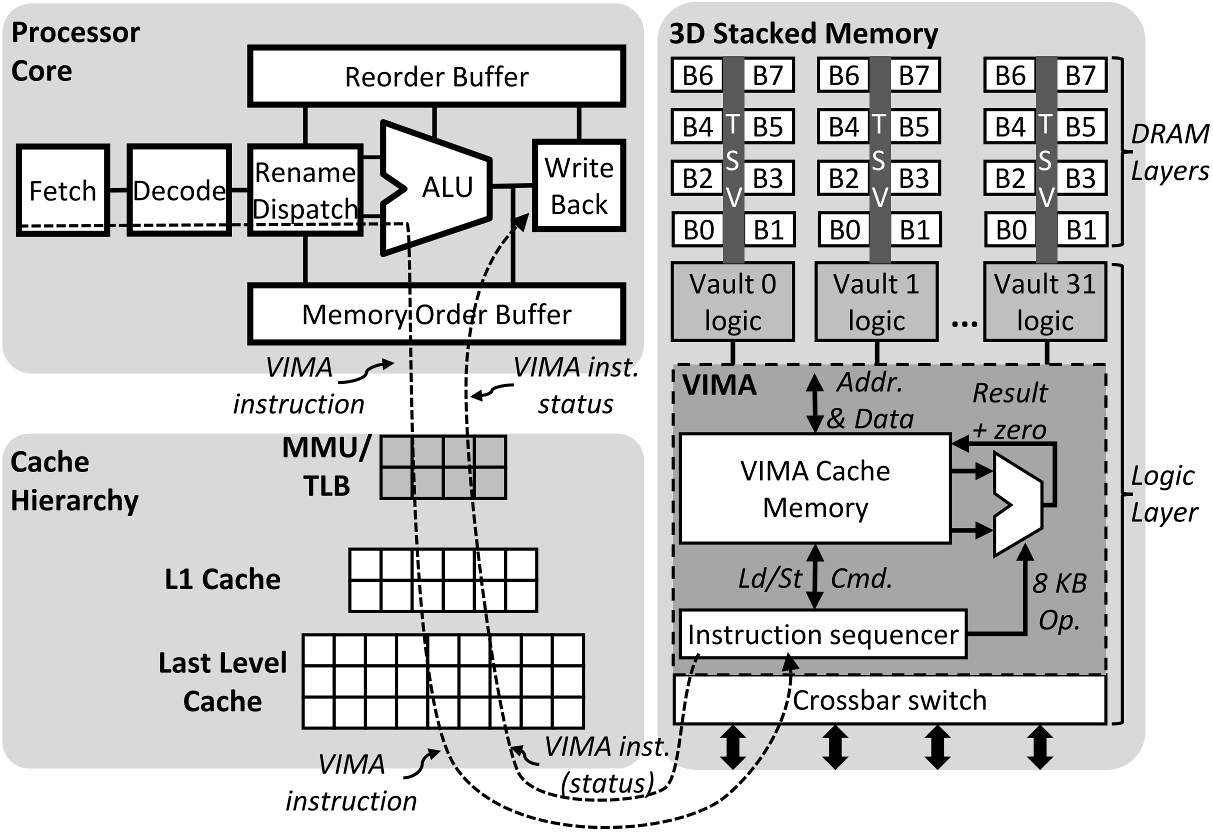


Figura 3.1: 3D-stacked memory module with the VIMA architecture.

Although VIMA instructions pass through the pipeline like regular memory instructions, they still require an Instruction Set Architecture (ISA) extension. All VIMA instructions are placed inside the Memory Order Buffer (MOB) and sent into the memory system upon traversing the pipeline. Once the memory returns a signal informing the execution status of an instruction (similar to what happens with AVX instructions), the processor reacts to this signal by committing the instruction if it was successfully executed or otherwise flushing the pipeline, raising an exception, and taking the necessary steps to handle it.

The processor shall provide precise exceptions by dispatching one VIMA instruction at a time (it dispatches the next VIMA instruction only after committing the last preceding one). This stop-and-go leads to two sources of performance reduction. First, since there is parallelism in the execution of VIMA instructions, when VIMA uses smaller vector sizes, it does not fully use the memory's internal parallelism. For instance, VIMA using 256 B vectors performs, on average, 74% worse than an equivalent version using 8192 B vectors. The second impact of precise exceptions is an execution gap between instructions. The impact of such pipeline bubbles is small for VIMA, varying between 2% and 4%. Every VIMA instruction generates at least one load or store operation into the memory. Therefore, these memory addresses are translated by the Translation Look-aside Buffer (TLB) and go through permission checks like any memory operation. We assume hardware support for large TLB pages, a common feature in modern processors (Kwon et al., 2016). VIMA instructions bypass the cache hierarchy. The memory addresses touched by any VIMA instruction are written back from the system's cache hierarchy to the main memory before execution to guarantee cache coherence. The coherence protocol must be VIMA-aware. It must write-back dirty lines and invalidate cache belonging to the pages on which VIMA will operate. These instructions then move into the VIMA instruction sequencer inside the 3D-memory, where the actual data access happens.

VIMA requires adding three elements to the 3D-memory: a set of vector functional units, an instruction sequencer, and a cache memory. This paper considers a memory with 32 vaults, 8 independent banks per vault, and 256 B row buffer size, but other layouts are also feasible. VIMA operates over vectors of 8 KB (32× 256B) utilizing the vault parallelism. One instruction can operate over 2048× 32-bit elements (e.g., integer) or 1024× 64-bit elements (e.g., floating-point). We used 256 parallel vector units, which means that eight extra cycles are required to fully process the 2048 elements in a pipelined fashion. This decision reduces the number of wires between the VIMA cache and the vector units.

VIMA executes instructions in-order. Instructions execute inside VIMA as soon as the required data is fetched from the memory vaults and made available in the VIMA's cache memory. If the data is available in the cache, 1 cycle is required for a tag-check, while another 8 cycles are required for 8 data transfers. These transfers and the functional units are fully pipelined, enabling complete parallelism. As most functional units require two operands, our design uses 2 cache ports to complete the operation in 8 cycles (required by the data transfers) plus the number of cycles required by the last pipelined operation.

Once an instruction finishes executing, a status signal is sent to the processor regarding completion or exception (similar to x86 AVX instructions). Before the execution, the sequencer checks the cache for the data that is required by each instruction. In case of a cache hit, the operation starts immediately. The operation ends by writing the results into a fill buffer. When VIMA sends a signal to the processor, it also writes from this buffer into the cache. This effectively hides the write operation inside the gap created by the "stop-and-go"approach. As we write one entire VIMA cache line at once, no Read-to-Modify operation is required. Whenever the VIMA cache evicts this dirty line, it will write it back to the main memory. During a miss, VIMA cache uses a Least Recently Used (LRU) policy to evict a line. VIMA cache splits each vector access into 128 sub-requests to the vault controllers (for 8 KB vectors, considering 64 B cache lines). Sub-requests guarantee minimal changes inside the DRAM devices as we are using the same cache line granularity. Besides, they are issued to different vaults and banks to increase parallelism.

3.2 EXPERIMENTAL EVALUATION OF VIMA

We used OrCS, an in-house cycle-accurate simulator, a simplified version of SiNUCA (Alves et al., 2015), to evaluate the architecture. The simulation parameters, which are described in

Tabela 3.1: 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;
	4 GB; DRAM@1666 MHz; 4-links@8 GHz; Inst. lat. 1 CPU cycle
	8 B burst width at 2.5:1 core-to-bus freq. ratio; Closed-row policy;
	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;
	256 int. units: alu, mul. and div. (8-12-28 cycles for 8 KB pipelined)
	256 fp. units: alu, mul. and div. (13-13-28 cycle for 8 KB pipelined);
	VIMA cache: 64 KB (8 lines), fully assoc., 2-cycle (1-tag, 1-per data);
	Dynamic energy: 194pJ per line access; Static power: 134mW;

Table 3.1 are similar to those of Intel's Sandy Bridge microarchitecture, with advancements such as large vector instructions (i.e., AVX-512).

We used 6 integer and floating-point kernels as workload. The integer kernels are *Memory Copy* and *Memory Set*, which generate most of the data movement in big data applications and typical consumer workloads (Boroumand et al., 2018). The floating point kernels are *Vector Sum*, *Matrix Multiplication*, *k-Nearest Neighbors*, *Multi-Layer Perceptron*, and *Stencil* convolution, which represent applications like neural network and computational fluid dynamics processing. For all applications (except *MatMul*), we used data sets of 4 MB, 16 MB, and 64 MB. We obtained the application traces using Pin (Bach et al., 2010) tool, aided by the VIMA Intrinsics tool (see section 5.5.2). Here is the description for each application:

- **MemSet**: sets all positions of a vector to a specific value.
- **MemCopy**: copies the contents of a vector to a new vector in a different memory location.
- **VecSum**: sums up each element of two input vectors storing the result in an output vector.
- **Stencil**: convolution using a 5-points stencil over a matrix storing the result in an output matrix.

- **MatMult**: multiplies two square matrices and stores the results in an output matrix. Due to simulation time, we adopted up to 8 MB in size per matrix, resulting in a total memory footprint of 6 MB, 12 MB, and 24 MB.
- **kNN**: classifies 256 test instances in an n-dimensional space. We used K to equal 9, 32768 training instances, varying the number of characteristics (32, 128, 512).
- **MLP**: neural network inference step. We used 32768 test instances, varying the number of characteristics (64, 256, 1024).

Figure 3.2(a) shows the speedup results for the benchmarks using VIMA compared to AVX as baseline, while varying the input size. We only mention results for VIMA using 8 KB vectors, as this size enables the 32 vaults' parallelism.

Speedup on integer benchmarks *MemSet* and *MemCopy* happens mainly because of the superior use of parallelism in the memory when fetching data without any data reuse. Still, it is partially limited because each VIMA instruction generates only a single VIMA cache miss at a time. In contrast, whenever VIMA requires two operands (generating two vector misses), both are requested leveraging the bank parallelism inside each vault. VIMA presents a similar execution time for these two applications, while AVX presents faster execution for *MemSet*, which has a smaller memory footprint.

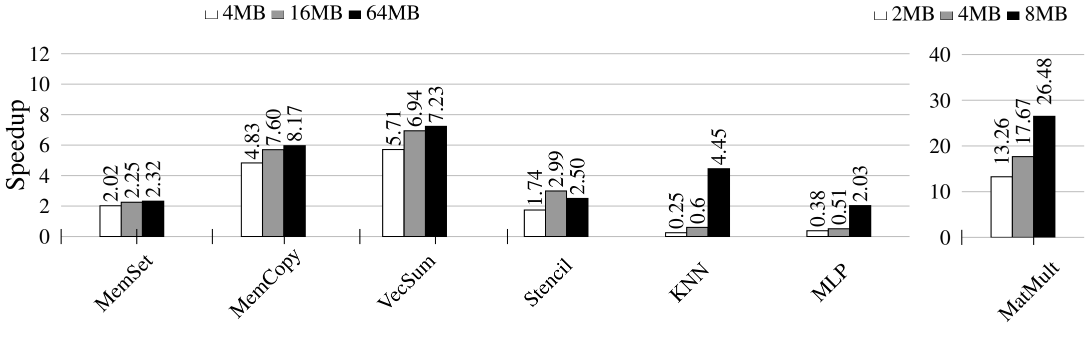


Figura 3.2: Speedup of VIMA normalized to baseline AVX with a single thread (higher is better).

The execution of *VecSum* using VIMA offers significant performance improvements by making good use of parallelism in the main memory. Namely, by fetching two large vectors in parallel, VIMA outperforms AVX by over 7× for this benchmark with the largest input size.

The *Stencil* algorithm offers good opportunities for reuse of vectorized data and thus also shows significant speedup. Here, data fetches with a single element stride are expected and can be served by the cache. Our baseline system considered a 16 MB last level cache. Results vary according to input size and matrix size, considering whether the dataset fits inside the last level cache of the baseline system and how efficiently the algorithm deals with different matrix dimensions. These factors cause the smaller speedup for the 4 MB input size and the increase of speedup between the 16 MB and 64 MB data sets.

Results for *kNN* and *MLP* using VIMA present no speedup whenever the data set used fits in the cache hierarchy of the baseline system (4 and 16 MB cases). For these cases, the processor cache hierarchy provides quick access to all the data necessary for processing. However, the speedup is considerable when the input size exceeds the size of the last level cache. VIMA is up to 4× faster than AVX for the 64 MB datasets, when the x86 cache presents no help.

The *MatMul* application uses a total of 6 MB, 12 MB, and 24 MB, considering the three matrices. For a straightforward, clear comparison of the memory access performance, we used the same algorithm for AVX and VIMA, which led to higher gains for VIMA. However, in our tests a tiled algorithm for AVX can result in up to 4× improvements. In such scenario, VIMA would still be over 6.5× faster for the 24 MB problem size.

In comparison to multithreaded AVX, our discussion considers only the largest sizes of benchmarks *Stencil*, *VecSum*, and *MatMult*. Figure 3.3 compares VIMA with an AVX implementation using up to 32 cores. The percentages above the plot indicate the energy consumption of each execution relative to AVX single-threaded execution, in the respective order.

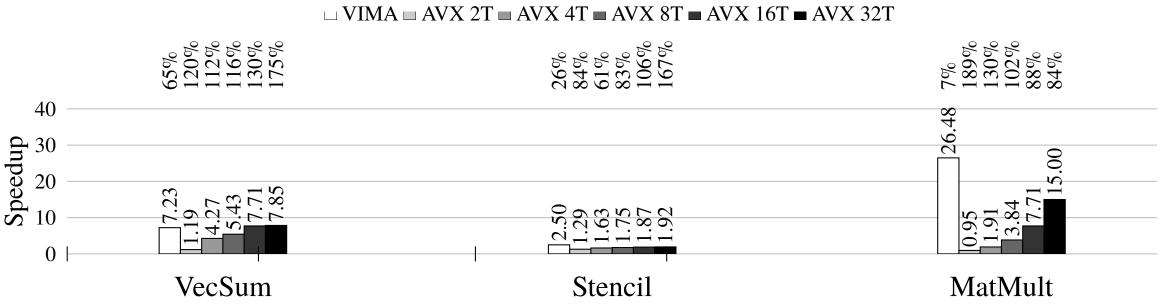


Figura 3.3: Speedup and energy of VIMA and AVX multithread normalized to baseline AVX with a single thread (higher is better). Numbers on the top of the plot present energy consumption relative to AVX single thread.

Considering these results, VIMA offers both superior performance and significant energy savings when compared to a single-threaded execution. It continues to outperform the baseline system for *VecSum* when these are executed with up to 16 cores in parallel, at a very small fraction of the consumption of energy. For *Stencil* and *MatMult* applications VIMA presents better performance even compared to AVX with 32 threads. For such applications VIMA benefits from the internal 3D-memory parallelism, while the VIMA cache provides necessary data reuse to enable gains. At the same time, VIMA does not rely on multiple cache levels, which would add extra latency to the memory latencies during a sequence of misses.

These results inform us of many desirable features of VIMA that make it interesting for our proposal. We believe we can leverage characteristics such as the efficient use of the parallelism of the 3D-stacked memory and the ability to process large amounts of data per instruction to design a solution that helps us achieve our goal. Thus, we consider VIMA as a basis for our proposal from this point onward.

4 RELATED WORK

Our research proposal's goal consists basically of leveraging the capabilities of a NDP architecture to implement a Bloom filter structure on hardware to aid applications that deal with large data sets with more efficient mechanisms for data fetching.

With this goal in mind, in this Chapter we discuss published research efforts found in the literature that relate to the main subjects of this document: Near-Data Processing and Bloom filters.

4.1 NEAR-DATA PROCESSING

The Near-Data Processing (NDP) capabilities of 3D-stacked memories have been explored by researchers in several distinct areas. Xie et al. (2017), for instance, move a portion of the computations necessary to render 3D images to the logic layer of a 3D-stacked memory. The authors aim to reduce data traffic during some of the more memory-intensive portions of graphics processing algorithms. Korikawa et al. (2020) use NDP in Network Function Virtualization (NFV) environments to speed-up packet processing by leveraging the bank interleaving and channel parallelism capabilities of 3D-stacked memories.

Numerous research efforts in NDP address issues brought about by requirements of big data applications, which process large data sets and are therefore particularly susceptible to the pitfalls of data movement and similar to Vector-In-Memory Architecture (VIMA). Lee et al. (2019) identify data redundancy in data centers and propose a NDP accelerator for inline Data Deduplication (DU) that significantly reduces latency and power consumption in comparison to traditional DU tools. NDP research efforts will sometimes include placing Accelerated Processing Units (APUs) or ARM cores on the logic layer of a 3D-stacked memory. These works focus on Machine Learning (ML) training functions (Gao et al., 2015), large-scale graph processing (Ahn et al., 2015), and in-memory network frameworks (Gao et al., 2015). These require ARM cores working in conjunction with a Translation Look-aside Buffer (TLB) and rely on routers for communication between vault. VIMA, on the other hand, is simpler and less costly, since it does not propose adding cores to the system and does not rely on vault communication to enhance performance.

Other proposals add specific-purpose cores to the 3D-stacked memory in a NDP architecture. For instance, NIM (Oliveira et al., 2017), a reconfigurable Neural Network (NN) accelerator, is similar to VIMA in that it is a NDP architecture that allows for vector operations, features processing units and a sequencer, and attaches to the crossbar switch. It is, however, much more complex and expensive as it requires one register bank per vault, while VIMA stores data in a cache that enables data reuse by design and is accessible from all vaults. Several similar efforts apply similar principles and offer features such as deactivating Functional Units (FUs) on demand and enhancing the execution of Convolutional Neural Networks (CNNs), but all of these require adding new modules for each vault of a 3D-stacked memory, which makes them more expensive and complex than VIMA.

Processing-In-Memory (PIM) techniques do not rely on 3D-stacked memories. They instead propose modifying conventional Dynamic Random Access Memory (DRAM) memories and repurposing some of its internal circuits to achieve computation capabilities (Gao et al., 2018; Deng et al., 2018; Li et al., 2017; Deng et al., 2019; Sim et al., 2018; Sudarshan et al., 2019). While this is generally not an expensive approach, it is a lot trickier to program and error-prone than VIMA as the programmer often must take care of low-level implementation details.

4.2 BLOOM FILTERS

In this section we discuss related work pertaining application domains that benefit from Bloom filters and a number of existing hardware implementations of Bloom filters found in the literature.

4.2.1 String Matching Applications

Consider a set of n strings $S = \{s_1, s_2, s_3, s_4, ..., s_n\}$. A string matching module observes an input stream with a shifting window of x characters. As the stream progresses, S is continuously queried with the x characters within the window being considered. The module must consistently and reliably return a result pertaining to whether these x characters match any string found in S. Naturally, this must also work when the queried data is composed of separate items, as opposed to a stream.

Due to how efficiently Bloom filters are able to represent and check for membership in sets of items, they are often used by applications that rely on string matching (Zengin and Schmidt, 2016). More specifically, because of their space efficiency and constant computation time, Bloom filters are commonly used to support applications that require real-time processing and/or must deal with large amounts of data. This can be observed in how widely Bloom filters are used in domains such as network applications (Broder and Mitzenmacher, 2004), antiviruses (Ho and Lemieux, 2008), database systems (Patgiri et al., 2019), and genome sequencing (Khairy et al., 2017).

Network applications must observe the data stream that arrives at a host through the network link and react according to the content. This is a difficult task, as increasing network speeds require high throughput from these applications lest they become processing bottlenecks. Bloom filters may be used to support several kinds of network applications, such as Network Intrusion Detection Systems (NIDS) (Nourani and Katta, 2007; Meghana et al., 2016), packet classification services (Nikitakis and Papaefstathiou, 2008), domain blocking (Yu et al., 2010), and IP address lookup (Byun et al., 2019).

NIDS monitor all network activity arriving at a host in order to identify and avoid possible intrusive or malicious behavior (Nourani and Katta, 2007). Most approaches to this involve trying to match strings present in the incoming stream with a list of signatures indicating known unwanted behavior. Anti-viruses, while functioning in a similar way to NIDS, are much more computationally intensive as they work with much larger sets of signatures that are also much longer. According to Ho and Lemieux (2008), anti-viruses may cause a 40% slowdown in boot time and up to 1000% in disk input/output performance. Bloom filters have also been used to mitigate some of this pressure.

Bloom filters are interesting for genome applications because of the size of the bases these applications must handle and due to their ability to map data of variable size into a fixed memory space. The input of such applications are DNA strands which must be matched with known DNA sequences of arbitrary size. This is the case of DNA sequencing algorithms like BLASTN, one of the fundamental bioinformatics algorithms (Khairy et al., 2017; Bhalekar and Chilveri, 2015). DNA strands are combinations of nucleotides (adenine, cytosine, guanine and thymine) and the sequencing process considers that similar sequences may indicate similar biological function. Since genomic sequences are very large, possibly requiring several billions of bases to be considered for sequencing, the string matching steps within algorithms like BLASTN and k-mer counting must be approached with clever solutions. Several implementations of these algorithms make use of Bloom filters in their word matching phases (Mcvicar et al., 2017).

Lastly, Bloom filters have also become widely used in Big Data applications and research. Big Data applications consider, by design, very large data sets. Due to their ability to deal with large sets in a space-efficient fashion and to check for item membership in constant time (Gupta and Batra, 2017), Bloom filters are very useful in this domain. They are used mainly by database applications aimed at reducing the number of items that must be considered for join operations (Lahiri et al., 2015; Lee et al., 2012; Pagare and Shinde, 2013), data deduplication tasks to reduce the size of data sets to be processed (Patgiri et al., 2019; Li et al., 2014) and generally sorting through large streams of data (Kaur and Sood, 2017; Lakshman and Malik, 2010).

4.2.2 Hardware Implementations

Since they are also relatively simple structures, there has been continued interest in developing hardware implementations of Bloom filters. These implementations are often used as bandwidth amplifiers or co-processing elements (Fang et al., 2020). Since the hash functions used by the structure must be independent from one another, they can be executed in parallel, which enables hardware implementations to operate very efficiently.

Many authors have focused on modeling and describing how Bloom filters can be implemented on hardware as a general purpose tool to support various kinds of applications. For instance, Zengin and Schmidt (2016) thoroughly analyze how a general purpose string matching module must work. The authors apply knowledge over the locality of the input data they consider and use a two filter design to achieve very high throughput and a much lower false positive rate than that of a conventional Bloom filter. They then describe how such an architecture can be implemented on hardware.

Khairy et al. (2017) describe a pipelined implementation of a Bloom filter aimed at architectures with very limited memory resources. They propose the use of high level synthesis tools to enable the hardware to be designed with high level languages like C, C++ and SystemC. Wada et al. (2018) provide a very thorough discussion of Bloom filters and describes a very efficient hardware implementation using rolling hash functions and multiple Bloom filter computation engines. The authors claim their Field Programmable Gate Array (FPGA)-based implementation of their design outperforms an Intel Core i7-6700K processor by 227× considering a sequential algorithm for the same task. Cho and Choi (2014) employ a hardware implementation to support general purpose key-value storing operations, which can be used for data deduplication, online multiplayer gaming and other internet services. The authors argue that such a design is preferrable to relational databases as it offers improved scalability.

Other researchers have explored using hardware Bloom filters as a means to achieve reduced power consumption through acceleration of processing and fewer memory accesses on devices with energy constraints. Lyons and Brooks (2009) describe such a design aimed at implementation of a Bloom filter on FPGA or an Application-Specific Integrated Circuit (ASIC) circuit for Internet of Things (IoT) devices.

Several works can be found regarding hardware implementations of Bloom filters for more specific purposes and applications. Nourani and Katta (2007), for instance, describe an architecture that uses a Bloom filter to support a NIDS. NIDS monitor all network activity arriving at a host in order to identify and avoid possible intrusive or malicious behavior. By leveraging dedicated hashing capabilities offered by the processor and some parallelism in the hashing required in the Bloom filter inquiry, the architecture achieves throughput of up to 100Gbps while checking for 16000 different string matches. Also in the field of network applications, Nikitakis and Papaefstathiou (2008) provide a FPGA-based implementation of a Bloom filter design they call 2sBFCE (Dual Stage Bloom Filter Classification Engine), which they use to classify network packets according to a set of rules. The design uses little memory space, handles networks rates of up to 5Gbps, and supports over four thousand classification rules.

Other hardware implementations of Bloom filters that support network applications include:

- Yu et al. (2010), which uses a FPGA-based implementation of a conventional Bloom filter to block access to host domains associated with pornographic content;
- Meghana et al. (2016), which employs a counting Bloom filter as a basis for a NIDS;
- and Byun et al. (2019), which uses a vectorized Bloom filter to support IP address lookup with a high throughput.

Ho and Lemieux (2008) employ a Bloom filter variant called Bloomier filter which, on top of indicating whether a match is found in the filter, also returns which pattern caused the match. This greatly speeds up the process of checking for false positives. Their implementation is used for virus scanning. Similarly, Sangeetha and Ramasubramanian (2015) control hardware signatures used to support parallel programming and otherwise control general resource usage in multicore systems with a hardware Bloom filter.

K-mer counting is a data reduction tool widely used in DNA sequencing algorithms to improve performance, reduce redundant memory accesses and remove errors (Pellow et al., 2017). However, due to the enormous data sets involved in genome sequencing tasks, even this efficient algorithm encounters issues due to the amount of data. Mcvicar et al. (2017) use an FPGA-based implementation of a counting Bloom filter to support K-mer counting while also taking advantage of the parallelism of a 3d-stacked memory. The resulting design achieves a speedup of up to 18× over a baseline software implementation.

Other works focus on implementations created to support BLASTN, another very common bioinformatics algorithm used in genome sequencing, a variant of popular sequence analysis tool Basic Local Alignment Search Tool (BLAST). Bhalekar and Chilveri (2015) use a FPGA to implement a parallel Bloom filter and a near-perfect hashing strategy that supports the word matching phase of the BLASTN algorithm. The resulting design is reportedly computationally efficient as it drastically reduces the amount of data the algorithm must consider and avoids most of the processing redundancy usually associated with the task. Khairy et al. (2017) also focus on BLASTN and describes a FPGA-based implementation of a parallel pipelined partitioned Bloom filter that aids the word matching of the algorithm. The design effectively removes irrelevant genes from the data set, thus reducing the search space the algorithm must consider and achieving improved performance over a baseline software implementation.

Table 4.1 lists all research papers that were consulted for studying the subject of Bloom filter usage. The information listed pertains to title, application (when applicable) and whether the paper is a survey or a description of an implementation in hardware or software.

Tabela 4.1: Table of papers.

Authors	Paper title	Type	Application
(Broder and Mitzenmacher, 2004)	Network applications of Bloom filters: A survey	Survey	Network applications
(Nourani and Katta, 2007)	Bloom filter accelerator for string matching	Hardware	General string matching
(Ho and Lemieux, 2008)	PERG: A scalable FPGA-based pattern-matching engine with consolidated Bloomier filters	Hardware	Systems security
(Nikitakis and Papaefstathiou, 2008)	A memory-efficient FPGA-based classification engine	Hardware	Network packet classification
(Lyons and Brooks, 2009)	The design of a Bloom filter hardware accelerator for ultra low power systems	Hardware	Low power devices
(Yu et al., 2010)	Blocking pornographic, illegal websites by internet host domain using FPGA and Bloom filter	Hardware	Host domain blocking
(Lakshman and Malik, 2010)	Cassandra - A Decentralized Structured Storage System	Software	Big Data Processing
(Lu et al., 2011)	A forest-structured Bloom filter with flash memory	Hardware	Data deduplication
(Lee et al., 2012)	Join Processing Using Bloom Filter in MapReduce	Software	Database systems
(Pagare and Shinde, 2013)	Recommendation System using Bloom Filter in MapReduce	Software	Recommendation systems
(Fan et al., 2014)	Cuckoo filter: Practically better than Bloom	Software	General purpose
(Li et al., 2014)	Secure deduplication storage systems with keyword search	Software	Data deduplication
(Cho and Choi, 2014)	An FPGA implementation of high-throughput key-value store using Bloom filter	Hardware	Database systems
(Bhalekar and Chilveri, 2015)	A review: FPGA based word matching stage of BLASTN	Hardware	DNA sequencing
(Lahiri et al., 2015)	Oracle Database In-Memory: A Dual Format In-Memory Database	Software	Database systems
(Sangeetha and Ramasubramanian, 2015)	A survey of hardware signature implementations in multi-core systems	Survey	Hardware signature systems
(Meghana et al., 2016)	SoC implementation of network intrusion detection using counting Bloom filter	Hardware	Network intrusion detection
(Zengin and Schmidt, 2016)	A fast and accurate hardware string matching module with Bloom filters	Hardware	General string matching
(Mcvicar et al., 2017)	K-mer counting using Bloom filters with an FPGA-attached HMC	Hardware	Genome sequencing
(Gupta and Batra, 2017)	A short survey on Bloom filter and its variants	Survey	General purpose
(Khairy et al., 2017)	Bloom filter acceleration: A high level synthesis approach	Hardware	General purpose
(Kaur and Sood, 2017)	Efficient resource management system based on 4vs of big data streams	Software	Big Data Processing
(Pellow et al., 2017)	Improving Bloom filter performance on sequence data using k-mer Bloom filters	Hardware	Genome sequencing
(Wada et al., 2018)	Efficient byte stream pattern test using Bloom filter with rolling hash functions on the FPGA	Hardware	General purpose
(Patgiri et al., 2019)	Role of Bloom filter in big data research: A survey	Survey	Big data
(Byun et al., 2019)	Vectored-Bloom filter implemented on FPGA for IP address lookup	Hardware	IP address lookup
(Santos et al., 2020)	Um Modelo de Indexação e recuperaAção privada de Documentos	Software	Data storage

5 PROPOSAL

In this chapter, we present our proposal, establish a hypothesis, set objectives, and discuss our methodology for reaching such objectives and thus prove our hypothesis.

5.1 HYPOTHESIS AND OBJECTIVES

Looking to improve the performance and energy efficiency of computer applications that use Bloom filters for checking data items for membership of a set, we formulate the following hypotheses:

- It is feasible to implement a Bloom filter structure on a NDP-enabled 3D-stacked memory device;
- Such an implementation can be used by applications for membership check operations;
- Such an implementation will yield faster execution of applications that take advantage of it;
- Such an implementation will yield reduced energy consumption by applications that take advantage of it;

For motivation, we consider an application that uses a software implementation of a Bloom filter to check numerous input data items for membership in an extensive data set. If one considers execution on a regular x86 architecture, the task is completed by running Algorithm 1. Considering, for instance, the baseline system described in Table 3.1 most memory requests yield a latency of several hundreds of cycles by checking all cache levels and accessing the main memory to locate each 64 B block of input data and subsequently transfer it back to the processor, where the verification of each item produces additional latency.

Algorithm 1: Membership check with bloom filter

```
Result: A list of data items that belong to the set
load Bloom filter from memory
while there are unchecked items do
    load next item from memory
    check Bloom filter for membership
    if Bloom filter result is positive then
        add item to result list
    end
end
return list of data items belonging to the set
```

In contrast, if one considers a hardware implementation proposal of the same Bloom filter, improvements in performance may come from a number of sources: i) When a VIMA instruction must access modified data within the cache hierarchy, to guarantee cache coherence, this data is written back to the memory before the instruction can be executed. However, when the data is not in the cache, VIMA avoids the latency of checking the cache hierarchy by bypassing it completely. This bypass should contribute to better overall performance when we already know that the data is not in any of the cache levels. This behavior is particularly advantageous when the data set's size exceeds the last level cache size. In this situation, a traditional architecture will experience a high cache miss rate, which an NDP approach will avoid, therefore saving time and energy. ii) Since all Bloom filter membership checks would be executed in the memory, only possible false positives would have to be handled by the processor, meaning much less data is moved between memory and processor, which should contribute to decreasing the overall energy consumption by the system. iii) The use of vector operations allows both for better use of parallel access to data within the memory and for processing a large number data items per single operation, as VIMA is capable of operating over 8192 B vectors, which should contribute to performance and efficient utilization of resources.

In light of these hypotheses and motivation, **our main objective is to provide a hardware implementation of a Bloom filter that can be used by applications that deal with large data sets and can benefit from an efficient hardware implementation to aid this task.** We believe that, by leveraging the capabilities of a Near-Data Processing (NDP) architecture, we can provide an accelerator that achieves both improved processing speed and lower energy consumption. By providing such an accelerator, as the task relies on this new element for processing, rather than on a processor core, we wish to help mitigate performance limitations when processing large amounts of data caused by issues such as the von Neumann bottleneck and the memory wall.

5.2 INITIAL DESIGN

We have devised an initial idea considering the capabilities of Vector-In-Memory Architecture (VIMA). While this design is still incomplete, it should illustrate one of the possibilities for implementation of a Bloom filter structure considering this type of architecture.

This initial design considers support string matching applications, which is in keeping with most common uses of Bloom filters found in the literature, but neither our proposal nor this specific algorithm are limited to any one type of data. String hashing functions are defined generically by Ramakrishna and Zobel (1997) as Algorithm 3. This algorithm considers a string $s = c_1, c_2, ..., c_m$ of m characters, a seed v and a resulting hash h generated after subsequent transformations considering each character of s.

The initial value of h is generated by an *init* function that is dependent of seed v and subsequently transformed by a $step$ function that considers the current state of h and the next

Algorithm 2: Generic String Hashing Function

```
$h_0 \leftarrow$ init(v)
for each character $c_i$ in $s$ do
    set $h_i \leftarrow$ step (i, $h_{i-1}$, $c_i$)
end
return h $\leftarrow$ final ($h_m$, $v$)
```

character c_i of the string being hashed. When all characters have been consumed, a $final$ function generates the final h considering the last internal state h_m and the original seed. By defining $init$, $step$ and $final$, one sets a string hashing function.

We have chosen to implement a shift-add-xor hash function, due to both its simplicity and its efficacy (Ramakrishna and Zobel, 1997). This is illustrated by its use on related work that employs the function as part of a hardware Bloom filter implemented on Field Programmable Gate Array (FPGA) (Mcvicar et al., 2017). The function defines $init$, $step$ and $final$ as follows:

- $init(v) = v$
- $step(i, h, c) = h \oplus (L(h) + R(h) + c)$
- $final(h, v) = h||T$

This definition of the *init* function refrains from applying any transformations to the seed. The $step$ function uses *shift* (denoted by functions L and R), *add*, and *exclusive or* operations to modify each intermediate h_i and the $final$ function truncates the last internal value of h considering size constraint T to generate the result.

VIMA uses NEON ARM functional units for processing and so we have access to several instructions designed to support vector operations. Thus, to implement such an algorithm for the VIMA architecture, we first use the VLD instructions from the NEON ARM Instruction Set Architecture (ISA) to split strings into individual characters. This operation loads 64-bit chunks of data into NEON registers with optional deinterleaving, allowing data splits of different sizes. There are four distinct VLD variants available in the ISA, each pertaining to a different deinterleaving option (ARM, 2020):

- VLD1 is the simplest option, which applies no deinterleaving and thus simply loads 64 bits of data into a register as stored in the memory;
- VLD2, which deinterleaves data into odd and even elements in two separate registers;
- VLD3, which does the same for three registers;
- and VLD4, for four registers.

Figure 5.1 shows an example of the results of this deinterleaving on RGB image data. Each of the instructions with this separation behavior also allow us to configure how many bits

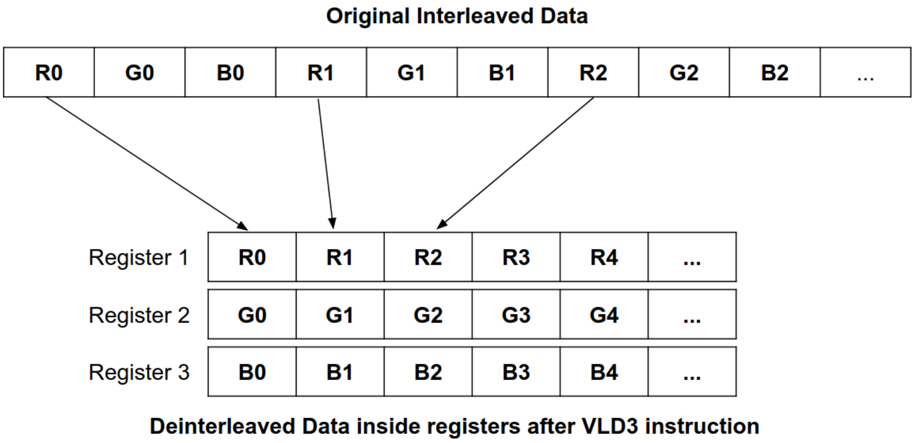


Figura 5.1: Data deinterleaving with the VLD operation.

must be considered per element, with 8-, 16- and 32-bit options. Considering this information, we have devised a Bloom filter design that leverages the capabilities of the architecture with which we are currently working and enables possibilities for further research on this topic. We propose a modification such that, when using VLD instructions from the NEON ISA, we are able to place results inside the VIMA cache, thus allowing the system to deinterleave data into multiple distinct VIMA vectors. Considering 8-bit characters, this enables our implementation to use the VLD4 instruction with 8-bit deinterleaving, thus placing characters separately in vectors. Since the instruction loads 64 bits of data, every load will transfer eight characters of a string, deinterleaving them in four vectors in a round-robin pattern.

If we assume 8KB VIMA vectors of 32-bit data points, we are able to hold in cache the first eight characters of up to 1024 strings, with two characters of each string per vector. To perform the *shift-add-xor* hash function would require then generating an initial 32-bit value placing it in every index of another vector. This vector will house every intermediate state of the hash. Two more vectors are necessary for the *shift-left* and *shift-right* operations used in the algorithm. Figure 5.2 illustrates this set of vectors. At this point, the algorithm can be performed with existing VIMA vector operations, ending with two distinct hash values for each string as a result. These values can then be used to address parallel hash tables or otherwise processed for any desired intent. Algorithm 3 describes this vectorized version of the *shift-add-xor* hash function.

Algorithm 3: Vectorized shift-add-xor Hashing Function

```
h ← init
for each character vector $v_i$ do
    $aux_1 \leftarrow L(h)$
    $aux_2 \leftarrow R(h)$
    $aux_1 \leftarrow aux_1 + aux_2$
    $aux_1 \leftarrow aux_1 + v_i$
    $h \leftarrow h \oplus aux_1$
end
```

S = {absolute, abstract, accepted, accurate, …, zambians}

h	*init*	*init*	*init*	*init*	*init*	*init*	…	*init*	*init*
v_1	a	l	a	r	a	p	…	z	i
v_2	b	u	b	a	c	t	…	a	a
v_3	s	t	s	c	c	e	…	m	n
v_4	o	e	t	t	e	d	…	b	s
aux_1									
aux_2									

Figura 5.2: Initial state of vectorized data for the shift-add-xor hash function. *h* holds all intermediate hash values, *v1-4* hold the string characters and *aux1-2* are used in the shifting operations.

When implementing a Bloom filter insertion or query operation, to set multiple independent hash functions it would suffice to set multiple initial 32-bit values for the hash vector and duplicate both these values and the string characters in their respective vectors as needed to achieve any number of distinct of independent hash functions. This works because every different initial hash value generates a different result and, thus, a distinct hash function. Figure 5.3 shows an example of such vectors using three hash functions. The algorithm stays the same, but since this requires some data redundancy, each vector holds characters of fewer strings.

S = {absolute, abstract, accepted, accurate, …, zambians}

h	*init1*	*init1*	*init2*	*init2*	*init3*	*init3*	…	*init3*	*init3*
v_1	a	l	a	l	a	l	…	a	r
v_2	b	u	b	u	b	u	…	c	a
v_3	s	t	s	t	s	t	…	c	t
v_4	o	e	o	e	o	e	…	u	e
aux_1									
aux_2									

Figura 5.3: Initial state of vectorized data for the shift-add-xor hash function. The three different initial values values in *h* mean three independent hash functions are computed.

The resulting vector holds in each data slot the address of a single bit in the Bloom filter that must be modified (for item insertion) or consulted (for inquiry). This could possibly be performed with *scatter* and *gather* operations, respectively, for instance. Since these operations are not yet available in the architecture we are considering, we have yet to devise how any remaining steps of the process will be performed. There is also the issue of, with this strategy, each string generating two hash values instead of one, which will also be considered and addressed in future steps of the research.

While we have considered string matching applications here, this approach can be applied to other data point sizes and thus be used for any other purpose. This initial design idea illustrates how the current capabilities of the VIMA design can be leveraged to implement a Bloom filter. We believe this design lays the groundwork on how any architecture with similar features can be implemented and may serve as a basis for our research into how such an implementation can support processing of any application that can benefit from Bloom filters.

We have yet to define a number of details that are important and shall be necessary for a functioning implementation of any kind, regarding Bloom filter configurations, architectural details and communication with applications. The following is a non-exhaustive list of such details:

- What size constraints will be considered for the Bloom filter;
- What data point sizes will be supported;
- How many independent hash functions will be considered;
- What details of the data structure will be adjustable by applications;
- Where the Bloom filter and/or any related operation results will be stored and accessed;
- How applications are going to access this solution;

All of these will be considered as we progress in the research. While our initial design considers the possibilities provided by VIMA, we foresee it may be necessary to modify the architecture by including additional components, adding new instructions to its ISA, or to propose an altogether new architecture.

5.3 RESEARCH PLAN

Here we discuss our methodology for achieving our main objective. We have devised a plan for the next steps of our research that we believe will be successful in confirming our hypothesis:

- **Consider Bloom filter variants**: as discussed in Section 4.2, there are many Bloom filter variants that deviate from the original idea as to modify the data structure and thus add features that may be desirable for various reasons. By studying these variants, we believe we will learn what features would be feasible and desirable for our eventual design.

- **Consider existing hardware implementations of Bloom filters**: we cited a number of existing hardware implementations in Section 4.2. We believe it will be useful to consider these successful cases as guides or references for our own implementation.
- **Propose an NDP implementation**: we believe at this point we will be able to propose an implementation that can act as a proof of concept. This implementation shall be designed within our in-house simulator, which we discuss on Subsection 5.5.1, which will also be used for testing and validation.
- **Analyze and select possible target applications**: after generating a proof-of-concept, we shall move onto which applications we can assist with our implementation. Once we learn more about the way these applications function and use their data, we believe will be able to select which ones we can efficiently support.
- **Adjust our NDP implementation according to selected target applications**: we predict our implementation will have to be somewhat tweaked in order to better assist the target applications.
- **Migrate target applications to use the proposed implementation**: by using the binary generation capabilities of our in-house simulator environment as will be discussed in Subsection 5.5.2, we plan to migrate the target application(s) so that we are able to test and validate our implementation.
- **Analyze obtained results in comparison with a baseline architecture**: we will then compare our results with those of an equivalent implementation using a traditional x86 architecture.

5.4 PROPOSED SCHEDULE

We envision the following steps for the development of this research:
Activity 1: Study existing general-purpose Bloom filter variants.
Activity 2: Study existing hardware implementations of Bloom filters.
Activity 3: Propose a NDP implementation and create a proof of concept.
Activity 4: Write and submit a research paper with our proposal and partial results.
Activity 5: Analyze and select possible target applications.
Activity 6: Adjust NDP implementation considering optimal data usage and storage patterns for the target applications.
Activity 7: Migrate target applications to use the proposed implementation.
Activity 8: Analyze the obtained results compared to a baseline architecture.
Activity 9: Write and submit a research paper with the obtained results.
Activity 10: Write the thesis and prepare the presentation for defense.

We plan to submit the research papers to one of the following conferences/journals:

Tabela 5.1: Planning chart.

	2021												2022					
Activities	J	F	M	A	M	J	J	A	S	O	N	D	J	F	M	A	M	J
A1	■	■																
A2	■	■																
A3		■	■	■	■													
A4						■												
A5							■	■										
A6								■	■	■	■							
A7										■	■	■	■	■	■			
A8										■	■	■	■	■	■			
A9																■		
A10																	■	■

- International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD);
- International Conference on Parallel Architectures and Compilation Techniques (PACT);
- International Symposium on Computer Architecture (ISCA);
- International Symposium on Circuits and Systems (ISCAS);
- Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD);
- International Symposium on High Performance Computer Architecture (HPCA);
- International Symposium on Microarchitecture (MICRO);
- Design, Automation and Test in Europe Conference and Exhibition (DATE);
- International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS);
- CCF Transactions on High Performance Computing;
- Advances in Electrical and Computer Engineering;
- Journal of Systems Architecture;
- IET Computers and Digital Techniques;
- ACM Journal on Emerging Technologies in Computing Systems;
- ACM Transactions on Architecture and Code Optimization;

5.5 METHODOLOGY

In this section we present the main tools that we plan to use for the evaluation of our proposal.

5.5.1 Ordinary Computer Simulator (OrCS)

Ordinary Computer Simulator (OrCS) is our in-house simulator, a simplified version of Simulator of Non-Uniform Cache Architectures (SiNUCA) (Alves et al., 2015), and it is a trace-driven simulator based on the x86 architecture. Since it is a trace-driven simulator, as opposed to a full-system simulator, it does not simulate the execution of instructions, but rather considers the overall behavior of the architecture components and latencies. The traces it uses are low-level descriptions of the execution of programs and they can be obtained through the use of binary instrumentation tools or created manually.

OrCS includes a tracing application that uses the Pin tool from Intel (Bach et al., 2010) to generate application traces compatible with the simulator. The application is analyzed by the tool during execution and corresponding trace files are generated considering the ISA supported by the simulator.

OrCS traces are composed of three separate files pertaining to distinct aspects of the execution of a program:

- **Static trace**: describes the binary of the application in assembly instructions organized in basic blocks. Each instruction includes an instruction identifier, opcode type, instruction address and size, read and write registers count and numbers, base and index registers, memory access addresses (if applicable), etc.

- **Dynamic trace**: stores an ordered list of basic block numbers that makes up the control flow of the execution the trace describes. Basic block identifiers consider the numbering defined in the static trace.

- **Memory trace**: lists all memory addresses referred to by the program during the entire execution. Each line includes whether the operation was a read or a write, what address it refers to, how much data was requested and in what basic block the operation happens.

5.5.2 Binary Generation

VIMA is composed by a series of vector units providing a vector ISA based on ARM NEON vector instructions (ARM, 2020). To program using the VIMA ISA we developed Intrinsics-VIMA, a library based on Intel and ARM intrinsics (Lomont, 2011) available in C and C++ language. The intrinsic libraries enable low-level code optimization through routine calls written in assembly. When the program calls any of these routines, the compiler embeds their Single Instruction Multiple Data (SIMD) instructions directly into the assembly code (Coorp., 2009; Cordeiro et al., 2017). Intrinsics-VIMA routines provide signed and unsigned operations represented in 32 and 64-bit for integer and floating-point single and double precision representations. Whenever our simulator finds an Intrinsics-VIMA routine during execution, it replaces with the equivalent VIMA instruction.

Previous work that integrates vectors near memory requires programmers to fine-tune the code to use the available registers or carefully allocate data inside specific memory banks and vaults. Intrinsics-VIMA simplifies the programming task by allowing developers to include VIMA function calls on the main code, as it works with any C and C++ library.

REFERÊNCIAS

Ahn, J., Hong, S., Yoo, S., Mutlu, O., and Choi, K. (2015). A scalable processing-in-memory accelerator for parallel graph processing. In *Int. Symp. on Computer Architecture*.

Al Hasib, A. (2018). Energy efficient computing on multi-core processors: Vectorization and compression techniques.

Alves, M. A. Z., Diener, M., Santos, P. C., and Carro, L. (2016). Large vector extensions inside the hmc. In *Design, Automation & Test in Europe Conf.*

Alves, M. A. Z., Villavieja, C., Diener, M., Moreira, F. B., and Navaux, P. O. A. (2015). Sinuca: A validated micro-architecture simulator. In *Int. Conf. on High Performance Computing and Communications*.

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

ARM (2020). Arm cortex-a57 technical reference manuals. `http://infocenter.arm.com/help/topic/com.arm.doc.subset.cortexa.a57/index.html`.

Bach, M., Charney, M., Cohn, R., Demikhovsky, E., Devor, T., Hazelwood, K., Jaleel, A., et al. (2010). Analyzing parallel programs with pin. *Computer*, 43.

Balasubramonian, R., Chang, J., Manning, T., Moreno, J. H., Murphy, R., Nair, R., and Swanson, S. (2014). Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34.

Bhalekar, S. R. and Chilveri, P. (2015). A review: Fpga based word matching stage of blastn. In *2015 International Conference on Pervasive Computing (ICPC)*, pages 1–4. IEEE.

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

Broder, A. and Mitzenmacher, M. (2004). Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509.

Byun, H., Li, Q., and Lim, H. (2019). Vectored-bloom filter implemented on fpga for ip address lookup. In *2019 International Conference on Electronics, Information, and Communication (ICEIC)*, pages 1–4. IEEE.

Chang, K. K. (2017). Understanding and improving the latency of dram-based memory systems. *arXiv preprint arXiv:1712.08304*.

Cho, J. M. and Choi, K. (2014). An fpga implementation of high-throughput key-value store using bloom filter. In *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*, pages 1–4. IEEE.

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

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

Deng, Q., Jiang, L., Zhang, Y., Zhang, M., and Yang, J. (2018). Dracc: a dram based accelerator for accurate cnn inference. In *Design Automation Conf.*

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 *Design Automation Conf.*

Ebrahimi, E., Mutlu, O., Lee, C. J., and Patt, Y. N. (2009). Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 316–326.

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. (1999). Computational ram: Implementing processors in memory. *IEEE Design & Test of Computers*, 16.

Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., and Burger, D. (2011). Dark silicon and the end of multicore scaling. In *Int. Symp. on Computer Architecture*.

Fan, B., Andersen, D. G., Kaminsky, M., and Mitzenmacher, M. D. (2014). Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88.

Fang, J., Mulder, Y. T., Hidders, J., Lee, J., and Hofstee, H. P. (2020). In-memory database acceleration on fpgas: a survey. *The VLDB Journal*, 29(1):33–59.

Gao, D., Shen, T., and Zhuo, C. (2018). A design framework for processing-in-memory accelerator. In *Int. Workshop on System Level Interconnect Prediction*.

Gao, M., Ayers, G., and Kozyrakis, C. (2015). Practical near-data processing for in-memory analytics frameworks. In *Int. Conf. on Parallel Arch. and Compilation*.

Ghose, S., Hsieh, K., Boroumand, A., Ausavarungnirun, R., and Mutlu, O. (2018). Enabling the adoption of processing-in-memory: Challenges, mechanisms, future research directions. *arXiv preprint arXiv:1802.00320*.

Gupta, D. and Batra, S. (2017). A short survey on bloom filter and its variants. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 1086–1092. 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*.

Ho, J. T. L. and Lemieux, G. G. (2008). Perg: A scalable fpga-based pattern-matching engine with consolidated bloomier filters. In *2008 International Conference on Field-Programmable Technology*, pages 73–80. 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`. [01-Jul-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/`.

Jacob, B., Wang, D., and Ng, S. (2010). *Memory systems: cache, DRAM, disk*. Morgan Kaufmann.

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

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

Kaur, N. and Sood, S. K. (2017). Efficient resource management system based on 4vs of big data streams. *Big data research*, 9:98–106.

Khairy, R., Safar, M., and El-Kharashi, M. W. (2017). Bloom filter acceleration: A high level synthesis approach. In *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–6. IEEE.

Korikawa, T., Kawabata, A., He, F., and Oki, E. (2020). Packet processing architecture using last-level-cache slices and interleaved 3d-stacked dram. *IEEE Access*, 8.

Kwon, Y., Yu, H., Peter, S., Rossbach, C. J., and Witchel, E. (2016). Coordinated and efficient huge page management with ingens. In *USENIX Symp. on Operating Systems Design and Implementation*.

Lahiri, T., Chavan, S., Colgan, M., Das, D., Ganesh, A., Gleeson, M., Hase, S., Holloway, A., Kamp, J., Lee, T.-H., et al. (2015). Oracle database in-memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1253–1258. IEEE.

Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40.

Lee, R. B. (1995). Realtime mpeg video via software decompression on a pa-risc processor. In *Digest of Papers. COMPCON'95. Technologies for the Information Superhighway*, pages 186–192. IEEE.

Lee, T., Kim, K., and Kim, H.-J. (2012). Join processing using bloom filter in mapreduce. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, pages 100–105.

Lee, Y. S., Kim, K. M., Lee, J. H., Choi, J. H., and Chung, S. W. (2019). A high-performance processing-in-memory accelerator for inline data deduplication. In *Int. Conf. on Computer Design*.

Li, J., Chen, X., Xhafa, F., and Barolli, L. (2014). Secure deduplication storage systems with keyword search. In *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, pages 971–977. IEEE.

Li, S., Niu, D., Malladi, K. R., Zheng, H., Brennan, B., and Xie, Y. (2017). Drisa: A dram-based reconfigurable in-situ accelerator. In *Int. Symp. on MicroArch.*

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

Lu, G., Debnath, B., and Du, D. H. (2011). A forest-structured bloom filter with flash memory. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6. IEEE.

Lyons, M. J. and Brooks, D. (2009). The design of a bloom filter hardware accelerator for ultra low power systems. In *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*, pages 371–376.

Mcvicar, N., Lin, C.-C., and Hauck, S. (2017). K-mer counting using bloom filters with an fpga-attached hmc. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 203–210. IEEE.

Meghana, V., Suresh, M., Sandhya, S., Aparna, R., and Gururaj, C. (2016). Soc implementation of network intrusion detection using counting bloom filter. In *2016 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pages 1846–1850. IEEE.

Mutlu, O., Ghose, S., Gómez-Luna, J., and Ausavarungnirun, R. (2019). Enabling practical processing in and near memory for data-intensive computing. In *Design Automation Conf.*

Nai, L. and Kim, H. (2015). Instruction offloading with hmc 2.0 standard: A case study for graph traversals. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 258–261.

Nikitakis, A. and Papaefstathiou, L. (2008). A memory-efficient fpga-based classification engine. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 53–62. IEEE.

Nourani, M. and Katta, P. (2007). Bloom filter accelerator for string matching. In *2007 16th International Conference on Computer Communications and Networks*, pages 185–190. IEEE.

Oliveira, G. F., Santos, P. C., Alves, M. A. Z., and Carro, L. (2017). Nim: An hmc-based machine for neuron computation. In *Int. Symp. on Applied Reconfigurable Computing*.

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*.

Pagare, R. and Shinde, A. (2013). Recommendation system using bloom filter in mapreduce. *International Journal of Data Mining & Knowledge Management Process*, 3(6):127.

Patgiri, R., Nayak, S., and Borgohain, S. K. (2019). Role of bloom filter in big data research: A survey. *arXiv preprint arXiv:1903.06565*.

Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., and Yelick, K. (1997). A case for intelligent ram. *IEEE Micro*, 17.

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

Pellow, D., Filippova, D., and Kingsford, C. (2017). Improving bloom filter performance on sequence data using k-mer bloom filters. *Journal of Computational Biology*, 24(6):547–557.

Ramakrishna, M. and Zobel, J. (1997). Performance in practice of string hashing functions. In *Database Systems For Advanced Applications' 97*, pages 215–223. World Scientific.

Sangeetha, R. and Ramasubramanian, N. (2015). A survey of hardware signature implementations in multi-core systems. In *2015 3rd International Conference on Signal Processing, Communication and Networking (ICSCN)*, pages 1–5. IEEE.

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

Santos, W. F., Felipe, C. H. F., and Viana, T. B. (2020). Um modelo de indexação e recuperação privada de documentos. *Revista Acta Kariri-Pesquisa e Desenvolvimento*, 2(1).

Schaller, R. R. (1997). Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59.

Sim, J., Seol, H., and Kim, L.-S. (2018). Nid: processing binary convolutional neural network in commodity dram. In *Int. Conf. on Computer-Aided Design*.

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 *Int. Symp. on Circuits and Systems*.

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`. [01-Jul-2019].

von Neumann, J. (1945). First draft of a report on the edvac. contract no. w-670-ord-4926. *The Origins of Digital Computers: Selected Papers, 3rd edn (Berlin/Heidelberg/New York: Springer-Verlag)*.

Wada, T., Matsumura, N., Nakano, K., and Ito, Y. (2018). Efficient byte stream pattern test using bloom filter with rolling hash functions on the fpga. In *2018 Sixth International Symposium on Computing and Networking (CANDAR)*, pages 66–75. IEEE.

Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23.

Xie, C., Song, S. L., Wang, J., Zhang, W., and Fu, X. (2017). Processing-in-memory enabled graphics processors for 3d rendering. In *Int. Symp. on High Performance Computer Architecture*.

Yu, H., Cong, R., Chen, L., and Lei, Z. (2010). Blocking pornographic, illegal websites by internet host domain using fpga and bloom filter. In *2010 2nd IEEE InternationalConference on Network Infrastructure and Digital Content*, pages 619–623. IEEE.

Zengin, S. and Schmidt, E. G. (2016). A fast and accurate hardware string matching module with bloom filters. *IEEE Transactions on Parallel and Distributed Systems*, 28(2):305–317.

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO - UFERSA
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO - PROPPG
Av. Francisco Mota, 572 – C. Postal 137 – Bairro Pres. Costa e Silva – Mossoró – RN – CEP: 59.625-900 - Tel.: (84)3317-8296/8295 – E.mail: proppg@ufersa.edu.br

RELATÓRIO DE AVALIAÇÃO DE DESEMPENHO

O aluno candidato a doutorado Sairo Raoní dos Santos tem desenvolvido sua pesquisa com grande entusiasmo e constancia. Mesmo em tempos de pandemia, o candidato tem mantido todos seus compromissos relacionados a pesquisa.

No último ano o candidato fez seu exame de qualificação o qual passou com sucesso.

Devido a alta carga de trabalho e o nível de concentração exigido ao longo do doutorado, considero que a continuidade de seu afastamento é fundamental para que o aluno possa alcançar seu objetivo de doutoramento com sucesso.

Reitero que mesmo durante a pandemia o candidato segue com reuniões diárias/semanais para orientação e compartilhamento com o grupo de pesquisa. Além disso, informo que sua presença física algumas vezes se faz necessária no laboratório para cuidar do gerenciamento de equipamentos usados em seus experimentos.

PROPPG

Data: 11 de abril de 2021

Prof. Marco A. Zanata Alves
Dep. de Informática
Mat.: 205129 - UFPR

Assinatura do(a) orientador (a)

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

DECLARAÇÃO DE MATRÍCULA

Data da emissão: 27/03/2021

Declaro para os devidos fins que SAIRO RAONÍ DOS SANTOS (CPF 01433292432), é aluno regularmente matriculado (matrícula número 201800110763) no curso de Doutorado do Programa de Pós-Graduação em INFORMÁTICA da UFPR, sob o número 40001016034P5. O referido aluno ingressou no Programa em 23/07/2018, com previsão para defesa da tese em 23/07/2022. Por ser verdade firmo a presente declaração.

Secretaria do Programa de Pós Graduação em
INFORMÁTICA

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

Nome: SAIRO RAONÍ DOS SANTOS		Documento: 01433292432
Data de Nascimento: 01/01/1990	Sexo: Masculino	Naturalidade: NÃO INFORMADO - Rio Grande do Norte
Filiação: Maria Elineuza dos Santos		

Curso: Doutorado em INFORMÁTICA	Data de Matrícula: 23/07/2018	Previsão de Titulação: 23/07/2022
Curso Reconhecido pelo Parecer nº 102/2011, D.O.U de 13/09/2012		
Área de Concentração: CIÊNCIA DA COMPUTAÇÃO	Linha de Pesquisa: Redes e Sistemas Distribuídos	
Lingua Estrangeira:		
Orientador: MARCO ANTONIO ZANATA ALVES		
Titulo da tese:		

HISTÓRICO ESCOLAR

Data de emissão: 27/03/2021

Código	Disciplina - Turma	CH/CR	Conceito	Per./Ano	Docente
Obrigatórias					
INFO-7042	SEMINÁRIOS EM INFORMÁTICA II - INFO - 7042	30/2	A	2° Sem./2018	ROBERTO PEREIRA
Eletivas					
INFO-7030	OFICINA DE SISTEMAS DISTRIBUÍDOS - INFO - 7030	60/4	9.0/A	2° Sem./2018	CARLOS ALBERTO MAZIERO, ELIAS PROCÓPIO DUARTE JÚNIOR, LUIS CARLOS ERPEN DE BONA
INFO-7005	ARQUITETURA DE COMPUTADORES - INFO - 7005	60/4	10.0/A	2° Sem./2018	MARCO ANTONIO ZANATA ALVES
INFO-7017	INTELIGÊNCIA ARTIFICIAL - INFO7017	60/4	9.5/A	1° Sem./2019	FABIANO SILVA
Validações de Créditos					
Código	Disciplina	CH/CR	Conceito	Crédito (Tipo)	Ano - Instituição
-	ENGENHARIA DE SOFTWARE	60/4	B	Eletivo (Aproveitamento)	2018 - UFERSA
-	INTELIGENCIA COMPUTACIONAL	60/4	B	Eletivo (Aproveitamento)	2018 - UFERSA
-	METODOLOGIA CIENTIFICA	30/2	A	Eletivo (Aproveitamento)	2018 - UFERSA
-	REDES DE COMPUTADORES	60/4	B	Eletivo (Aproveitamento)	2018 - UFERSA
-	SISTEMAS DISTRIBUÍDOS	60/4	A	Eletivo (Aproveitamento)	2018 - UFERSA
Creditos de Disciplinas para Titulação (necessários/concluídos) - Obrigatórios: 2/2 Eletivos: 34/30 Total: 36/32					

Resolução no. 32/17 - CEPE Conceito: A = Excelente (9.0 a 10.0) B = Muito Bom (8.0 a 8.9) C = Bom (7.0 a 7.9) D = Insuficiente (0.0 a 6.9)

Secretaria do Programa de Pós Graduação em

INFORMÁTICA

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO - UFERSA
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO - PROPPG
Av. Francisco Mota, 572 – C. Postal 137 – Bairro Pres. Costa e Silva – Mossoró – RN – CEP: 59.625-900 - Tel.: (84)3317-8296/8295 – E.mail: proppg@ufersa.edu.br

TERMO DE DECLARAÇÃO E COMPROMISSO

Eu, Sairo Raoní dos Santos, portador do CPF nº 014.332.924-32 RG nº 2459204, matrícula siape nº 2975474, devidamente autorizado(a) pela Universidade Federal Rural do Semi-Árido – UFERSA para realizar o curso de Doutorado no Programa de Pós-graduação em Informática da Universidade Federal do Paraná (UFPR), pelo presente e na melhor forma de direito, conforme a Lei nº 8.112/90, em seu Artigo 96-A, o Regimento Geral da UFERSA, em seu Artigo 338, e a RESOLUÇÃO CONSUNI/UFERSA Nº 003/2018, de 25 de junho de 2018, assumo o compromisso formal de permanecer, obrigatoriamente a serviço da UFERSA, por tempo integral e com dedicação exclusiva por um prazo igual ao do afastamento, a contar da conclusão do referido curso, sob pena de ressarcimento de todas as despesas, diretas ou indiretas em que a mesma tenha incorrido financiando aquele curso, tais como: salários, gratificações, passagens, diárias, ajudas de custo, bolsa de complementação salarial, bolsa de estudos, custos de matrícula, mensalidades e anuidades, enfim, qualquer dispêndio feito pela União, através da sua administração direta ou indireta, centralizada ou descentralizada, com o fim de custeio do curso em epígrafe.

Declaro estar ciente das Normas e Regulamentos do Curso.

Fica eleito o foro da Justiça Federal, Seção Judiciária do Rio Grande do Norte para dirimir todas as questões porventura decorrentes deste instrumento.

Curitiba (PR), 11 de abril de 2021.

SAIRO RAONI DOS SANTOS:01433292432
Assinado de forma digital por SAIRO RAONI DOS SANTOS:01433292432
Dados: 2021.04.12 08:38:11 -03'00'

Assinatura

Nome da testemunha: GRASCIELE FABIANA CASAGRANDE CENTENARO
CPF: 003749629009

Nome da testemunha: MARCO ANTONIO ZANATA ALVES -
CPF: 315 306 938 70

PROPPG