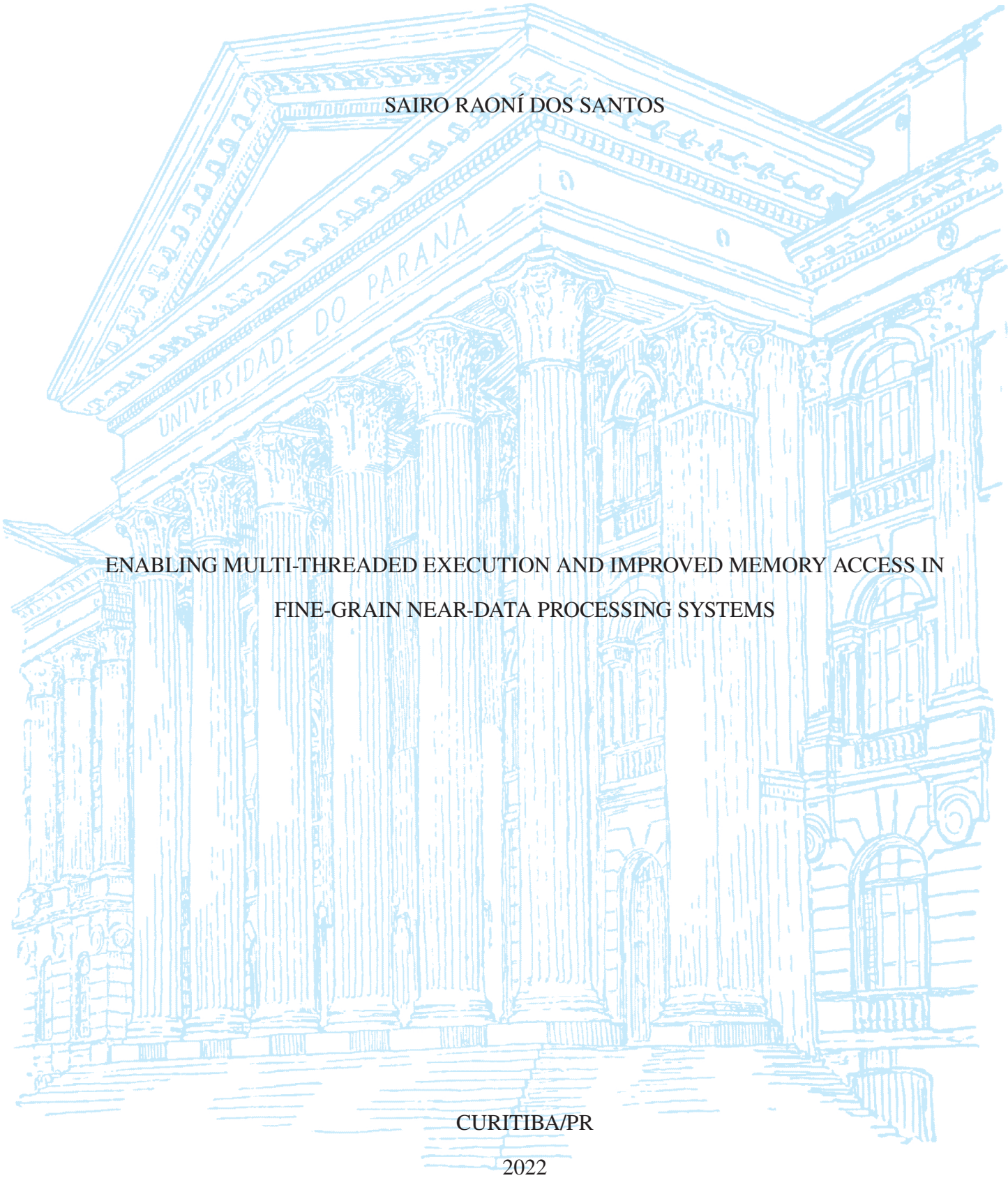
UNIVERSIDADE FEDERAL DO PARANÁ

SAIRO RAONÍ DOS SANTOS



ENABLING MULTI-THREADED EXECUTION AND IMPROVED MEMORY ACCESS IN FINE-GRAIN NEAR-DATA PROCESSING SYSTEMS

CURITIBA/PR

2022

SAIRO RAONÍ DOS SANTOS

ENABLING MULTI-THREADED EXECUTION AND IMPROVED MEMORY ACCESS IN FINE-GRAIN NEAR-DATA PROCESSING SYSTEMS

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

Área de concentração: *Ciência da Computação*.

Orientador: Marco Antonio Zanata Alves.

CURITIBA/PR

2022

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA CIÊNCIA E TECNOLOGIA

Santos, Sairo Raoní dos.
Enabling multi-threaded execution and improved memory access in fine-grain near-data processing systems. / Sairo Raoní dos Santos. – Curitiba, 2022.
1 recurso on-line : PDF.

Tese (Doutorado) – Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.
Orientador: Prof. Dr. Marco Antonio Zanata Alves.

1. Sistemas de memória de computadores. 2. Processamento eletrônico de dados. 3. Capacidade do computador. 4. Fluxo de dados (Computação). I. Alves, Marco Antonio Zanata. II. Universidade Federal do Paraná. Programa de Pós-Graduação em Informática. III. Título.

Bibliotecário: Nilson Carlos Vieira Júnior CRB-9/1797

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **SAIRO RAONÍ DOS SANTOS** intitulada: **ENABLING MULTI-THREADED EXECUTION AND IMPROVED MEMORY ACCESS IN FINE-GRAIN NEAR-DATA PROCESSING SYSTEMS**, sob orientação do Prof. Dr. MARCO ANTONIO ZANATA ALVES, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

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

CURITIBA, 08 de Julho de 2022.

Assinatura Eletrônica
11/07/2022 20:06:06.0
MARCO ANTONIO ZANATA ALVES
Presidente da Banca Examinadora

Assinatura Eletrônica
11/07/2022 19:10:12.0
MÔNICA MAGALHÃES PEREIRA
Avaliador Externo (UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE)

Assinatura Eletrônica
11/07/2022 19:25:55.0
EDUARDO CUNHA DE ALMEIDA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
20/07/2022 09:10:15.0
MARCIO SEIJI OYAMADA
Avaliador Externo (UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ)

RESUMO

Aplicações que lidam com grandes quantidades de dados são cada vez mais populares. No entanto, as arquiteturas tradicionais centradas em computação estão mal equipadas para lidar com essas aplicatções, pois elas causam muito movimento de dados no sistema devido aos acessos de dados quase constantes. Isso leva a um processamento ineficiente, com longos tempos de execução e alto consumo de energia. Os problemas causados por essa disparidade são amplamente conhecidos como *memory wall*. A partir do final da década de 1990, a ideia de mover parte da computação para perto da memória, quando benéfico, começou a ser considerada. Este conceito tornou-se conhecido como processamento próximo à memória e ganhou mais atenção no início da década de 2010 com o advento da tecnologia de Through-Silicon Via (TSV), que permitiu a integração direta das lógicas de processamento e armazenamento de dados no mesmo chip. Memórias 3D, que integram verticalmente armazenamento e lógica, tornaram-se comercialmente disponíveis desde então e pesquisadores da área de arquitetura de computadores reagiram propondo muitos projetos que colocam elementos de processamento na camada lógica normalmente encontrada nesses dispositivos. Esta tese propõe a Vector-In-Memory Architecture (VIMA), uma arquitetura de processamento próximo à memória baseada em memória 3D que implementa o processamento na memória colocando unidades funcionais na camada lógica desses dispositivos. Nosso projeto usa unidades funcionais vetoriais e uma memória cache para armazenamento dedicado e avança o estado da arte implementando exceções precisas e permitindo multi-threading próximo aos dadosna memória. Simulamos a execução de várias aplicações orientadas a dados em nossa arquitetura e, nossos resultados mostram que o design proposto, que utiliza 1 core e a VIMA, é capaz de superar uma arquitetura tradicional moderna de 16 cores em pelo menos 2× ao lidar com grandes tamanhos de conjuntos de dados. Além disso, essa aceleração no tempo de execução é alcançada enquanto se reduz o consumo de energia em pelo menos 75% de acordo com nossas estimativas. Em comparação com um trabalho similar do estado da arte, a VIMA é capaz de reduzir o tempo de execução de aplicações que fazem streaming de dados em pelo menos 32%.

Palavras-chave: processamento próximo à memória. memórias 3D. exceções precisas.

ABSTRACT

Applications that deal with large amounts of data are increasingly popular. However, traditional computation-centric architectures are ill-equipped to handle such applications as they cause much data movement across the system due to their near-constant data accesses. This leads to inefficient processing, with long execution times and high energy consumption. Issues caused by this disparity are widely known as the memory wall. Starting in the late 1990s, the idea of moving portions of the computations close to the memory when beneficial began to be considered. This concept has now become known as Near-Data Processing (NDP) and gained more attention in the early 2010s with the advent of TSV technology, which enabled straight-forward integration of processing logic and data storage in the same chip. 3D-stacked memories, which vertically integrate storage and logic, have become commercially available ever since and computer architecture researchers have reacted by proposing many designs that place processing elements on the logic layer typically found in those devices. This thesis proposes VIMA, a 3D-stacked memory-based NDP architecture that implements processing in the memory by placing Functional Units (FUs) on the logic layer of those devices. Our design uses a vector functional units and a cache memory for dedicated storage and advances the state-of-the-art by implementing near-data precise exceptions and enabling near-data multi-threading. We simulate execution of several common data-driven applications on our architecture and, out results show that the proposed design, with only a single processing core and VIMA, is able to outperform a modern 16-thread by at least 2× when dealing with large dataset sizes. Moreover, such a speedup in performance is achieved while reducing energy consumption by at least 75% according to our estimates. In comparison to its most closely related state-of-the-art work, VIMA is able to reduce the execution time of data-streaming applications by at least 32%.

Keywords: near-data processing. 3d-stacked memories. precise exceptions.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ASIC	Application-Specific Integrated Circuit
ALU	Arithmetic Logic Unit
AMC	Active Memory Cube
API	Application Programming Interface
AVX	Advanced Vector Extensions
BCDA	Bandwidth-Critical Data Analysis
C-RAM	Computational-RAM
CFD	Computational Fluid Dynamics
CGRA	Coarse-Grain Reconfigurable Array
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DCNN	Deep Convolutional Neural Network
DDR	Double Data Rate
DMA	Direct Memory Access
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
FP	Floating-point
FPGA	Field-Programmable Gate Array
FU	Functional Unit
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory
HIVE	HMC Instruction Vector Extensions
HIPE	HMC Instruction Prediction Extensions
HMC	Hybrid Memory Cube
IDE	Integrated Development Environment
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
JEDEC	Joint Electron Device Engineering Council
KNN	K-Nearest Neighbors
LLC	Last-Level Cache
LRU	Least Recently Used
McPAT	Multi-core Power, Area, and Timing
MCN	Memory Channel Network

MMU	Memory Management Unit
MVX	Memory Vector eXtension
MR	MapReduce
NDP	Near-Data Processing
NN	Neural Network
NoC	Network-on-Chip
NUMA	Non-Uniform Memory Access
OoO	Out-of-Order
OpenMP	Open Multi-Processing
PE	Processing Element
PHY	Physical Layer
PRIMO	Processing-In-Memory cOmpiler
ReRAM	Resistive random-access memory
ROB	Reorder Buffer
RVU	Reconfigurable Vector Unit
SDR	Single Data Rate
SIMD	Single Instruction Multiple Data
SiNUCA	Simulator of Non-Uniform Cache Architectures
SSD	Solid-State Drive
SRAM	Static Random Access Memory
STT-RAM	Spin-Transfer Torque Random-Access Memory
TLB	Translation Look-aside Buffer
TOM	Transparent Offloading and Mapping
TSV	Through-Silicon Via
VIMA	Vector-In-Memory Architecture

CONTENTS

1 INTRODUCTION

Advances in processor technology, with vector processing, a nearly constant increase in number of transistors, out-or-order execution and instruction-level parallelism, have translated directly into faster processing for several decades. However, the technology used for the main memory of most computer systems has not keep up with such advances (Chang, 2017), causing a gap between processing and memory access latency. 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).

Since all modern computers are based on the von Neumann architecture (von Neumann, 1945), the memory wall poses a serious issue: the von Neumann architecture requires that any instruction and 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); and 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.

The idea of moving computation close to the memory was initially proposed back in the late 1990s (Patterson et al., 1997; Elliott et al., 1999) but failed to gain traction at the time. It required integrating processing and data storage elements on the same chip, which was considered too challenging. By the early 2010s, however, it started to reemerge as a viable option with the advent of Through-Silicon Via (TSV) technology (Olmen et al., 2008) and subsequent commercial release of 3D-stacked memories (Hybrid Memory Cube Consortium, 2012; Association, 2013).

3D-stacked memories are a recent main memory design that stacks multiple 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 and logical division in several individual vaults (similar to memory channels), 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 Near-Data Processing (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.

A NDP design can still follow the von Neumann model by placing entire processors near the data. This approach, however, may increase complexity and may present area, power and thermal issues (Eckert et al., 2014). Another possible approach is to extend the architectural model by placing FUs near-data, which avoids some of these issues and allows processors to continue handling tasks they excel at, such as fetching and decoding complex instructions, predicting the outcome of branches, among other functions. While an FU-based approach curbs complexity and requirements of area and energy consumption, it is still able to provide adequate power to process large volumes of data.

Several proposals have used this approach and achieved significant results regarding execution time (Alves et al., 2016; Tomé et al., 2018), energy efficiency and data throughput performance. Nonetheless, most of them still implement designs that both pose barriers for adoption as they fail to maintain a sequentially consistent model.

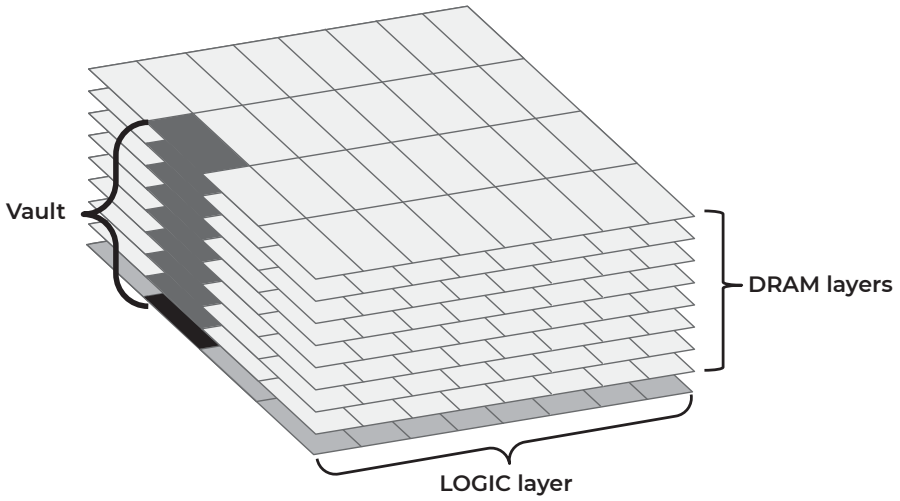


Figure 1.1: Block diagram of a 3D-stacked memory.

1.1 HYPOTHESIS

We pose the following hypothesis: **it is possible to provide precise exceptions, improved memory access and multi-threading with fine-grain NDP to speedup data streaming applications**.

1.2 CONTRIBUTIONS

Our main goal with this work is to contribute to the field of Near-Data Processing (NDP) with an optimization over existing state-of-the-art Single Instruction Multiple Data (SIMD)-based designs by guaranteeing precise exceptions while improving data throughput and enabling multithreading support. With our contributions, such architectures should be able to further speedup processing of data-streaming applications with low data reuse and a coalescent data access pattern.

To achieve this, in this thesis we propose Vector-In-Memory Architecture (VIMA). VIMA is an architecture that proposes adding vector units (i.e. SIMD) and a small cache memory to 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 (analogous to DDR channels). Along with its cache memory, we enable the short term data reuse, out-of-order instruction operand loading and near-data multithreading.

Our design includes contributions that advance fine-grain offloading NDP architectures by allowing them to efficiently extract data throughput from a 3D-stacked memory, which translates directly into improved performance for data streaming applications. Our simulation results indicate significant improvements in execution time and energy consumption when executing simple benchmarks such as memory set and copy operations and ubiquitous database queries.

We investigate a number of research questions and list our contributions as follows.

1. Can we design a NDP architecture that does not require coding in assembly and can be programmed without a specific compiler?

    - We propose VIMA, a SIMD-based near-data processing architecture that is placed on the logic layer of a 3D-stacked memory (Chapter 4).
    - We provide Intrinsics-VIMA, an intrinsics library that allows programmers to code and debug applications for VIMA on any system that supports C/C++ programming (Section 4.1).

    This work was completed and published on Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2021) (Cordeiro et al., 2021) and arXiv.org (Alves et al., 2022).

2. Is it possible to accelerate the execution of common machine learning algorithms with near-data execution?

    - We design, implement and simulate vectorized near-data versions of the k-Nearest Neighbors (kNN) and Multi-layer Perceptron (MLP) algorithms (Cordeiro et al., 2021).
    - We implement a vectorized version the stencil algorithm for simulated near-data execution and discuss our experimental findings (Chapter 5).

    This work was completed and published on Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2021) (Cordeiro et al., 2021).

3. What are the application domains best suited and most commonly migrated to near-data processing solutions?

    - We study the existing literature on near-data processing research and discuss the main aspects that impact the design of near-data processing architectures (Chapter 2).
    - We study the existing literature on near-data processing research and identify the most common application domains addressed in the field (Chapter 3).

    This work was completed and published on Journal of Integrated Circuits and Systems (Santos et al., 2021b).

4. Is it possible to accelerate the execution of common database query operators with near-data execution?

    - We design, implement and simulate the selection, projection and bloom join database query operators (Chapter 5).
    - We discuss our experimental findings of the simulated execution of our implementations (Chapter 5).

    This work was completed and published on Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2022) (Santos et al., 2022b).

5. Can we provide a near-data implementation of precise exceptions while optimizing utilization of memory resources and improving overall performance of data-streaming applications?

- We use VIMA to implement precise exceptions near the data while optimizing data throughput usage of the 3D-stacked memory (Chapter 4).
- We discuss various approaches to maintaining cache coherence in a NDP-enabled system, going over the overhead of each strategy.
- We present a comprehensive performance analysis of data-driven applications running on our architecture, comparing it to a 16-thread x86 baseline and an existing state-of-the-art NDP architecture (Chapter 5).

Partial results have been published at International Symposium on Performance Analysis of Systems and Software (ISPASS 2022) (Santos et al., 2022a), Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2022) (Santos et al., 2022b), and XXII Escola Regional de Alto Desempenho da Região Sul (Santos and Alves, 2022).

According to our experiments, VIMA is able to efficiently leverage the internal data throughput of 3D-stacked memories to provide improvements in overall system performance. When compared to a 16-thread x86 baseline, a single-thread system using our architecture proposal is able to provide better execution time performance with a speedup at least 2×, while consuming only a small fraction of the energy, across several data-driven applications simulated. Our results improve upon the most closely related state-of-the-art work by at least 32%.

1.3 THESIS OVERVIEW

The remainder of this text is structured as follows. Chapter 2 discusses background information on NDP research, going over the many possible technologies, architectures and programming models that have been used to implement NDP. Chapter 3 gives an overview of the related work available in the literature and also discusses the main domains most commonly targeted by NDP proposals. Chapter 4 describes our proposal, VIMA, in detail. Chapter 5 discusses the experimental results we have been able to achieve with VIMA, evaluating it against the performance of a 16-thread traditional architecture and an existing state-of-the-art NDP architecture. Chapter 6 concludes the text and mentions a number of possibilities for future work.

2 BACKGROUND

Near-Data Processing (NDP) is an approach to computation that moves processing close to the data, thus reducing data access times and energy consumption when processing data-intensive tasks. The concept of NDP extends the von Neumann architecture model by adding processing capabilities outside of the processor and near the memory, thus eliminating the need for some of the data movement between memory and processor that is required by the traditional model. Instead of moving massive amounts of data from the memory to the processor for processing, an NDP architecture can move only a few bits of data from the memory containing an instruction that will be then offloaded for near-data execution. When considering data-centric applications that constantly access data, using such an approach significantly reduces execution time and energy consumption at the same time as it exploits the parallelism and internal bandwidth of the main memory in systems.

Although the idea was originally proposed back in the late 1990s (Patterson et al., 1997; Elliott et al., 1999), it was considered impractical at the time since it required integrating data storage and logic cells on the same die: while Static Random Access Memory (SRAM) storage uses the same fabrication process as processors and can thus be easily integrated onto processing elements, Dynamic Random Access Memory (DRAM) storage, which is the technology most commonly used for main memory, is manufactured through a process that is sub-optimal for logic circuits (Jacob et al., 2010). With the advent of Through-Silicon Via (TSV) technology, which allowed for practical bundling of logic and DRAM layers, 3D-stacked memories caused interest in NDP to surge again. Designs such as the Hybrid Memory Cube (HMC) (Hybrid Memory Cube Consortium, 2014) and the High Bandwidth Memory (HBM) (Jun et al., 2017) have since become commercially available and been used as a basis for several NDP architectural proposals.

At the same time, development in memristor technologies has created another avenue for achieving in-memory computation, also yielding several architecture proposals capable of performing logic and arithmetic operations near the data (Lehtonen and Laiho, 2009; Taha et al., 2013). Other approaches even achieve a similar effect by modified the behavior of sense amplifiers in commercially available DRAM devices (et al., 2017).

In this chapter we describe and discuss several aspects that inform and impact near-data processing devices and architectures, such as the distance between device and data, the storage technology being used, the supported programming model and the code/task offloading mechanism.

2.1 MEMORY CELLS AND ARCHITECTURES

Different types of memories will usually lend themselves to different uses in a computer system so their capabilities can be exploited to their fullest extent. In general, NDP architectures try to access the data stored in the memory as directly as possible so as to improve bandwidth. By doing so, they are theoretically able to compute data in large portions without relying on buses for communication which, in turn, improves performance regarding both execution time and energy efficiency. Consequently, the memory architecture being used by a storage device directly impacts the possibilities of a near-data accelerator: the architecture informs how the data is organized, what data is stored, how the data can be accessed and also what it is used for in the context of the entire computer system.

The layout of memory cells in memory devices is usually designed to offer as much bandwidth as possible, in order to provide ease and speed of access to the stored data. In DRAM devices, this layout commonly arranges memory cells in a matrix, which are grouped to form banks. A device is then composed of a set of banks, from each of which stored data can be accessed by looking up specific rows in its matrices. Although Figure 2.1 illustrates a modern 3D-stacked design, it still abides by these organization guidelines, as do all modern memories. Every such device is made up of banks that can be accessed concurrently, which mitigates the latencies of the memory cells and provides parallelism in the data access. Such parallelism can be leveraged by clever data access behavior and policies to extract better performance.

SRAM-based NDP proposals, due to what this type of memory is commonly used for in computer systems, will usually place processing logic on cache hierarchies and memory controllers (Ando et al., 2017; Eckert et al., 2018; Wang et al., 2019; Yin et al., 2019; Ramanathan et al., 2020; Long et al., 2020; Aga et al., 2017; Sadredini et al., 2020; Cali et al., 2020; Hashemi et al., 2016; Awan et al., 2017). This means they will focus mostly on reducing data movement between the cache hierarchy and the processing core, thus improving execution time performance by freeing the host processor from a portion of the processing duties.

When considering traditional DRAM-based memories, such as the common 2D Double Data Rate (DDR) memory, several approaches have also been tried by NDP researchers. While some integrate full processing cores into the memory devices (Alian et al., 2018; Devaux, 2019), others implement customized circuitry to add SIMD capabilities to the memory (Alves et al., 2015b,a; Xi et al., 2015), or exploit internal access patterns to achieve in-memory processing (Li et al., 2017; Huangfu et al., 2020; et al., 2017; Gao et al., 2019; Xin et al., 2020; Hajinazar et al., 2021; Seshadri et al., 2018).

3D integration, however, has caused adding processing capabilities to memory devices to become much more straight-forward than before. 3D-stacked memories vertically connect several DRAM layers on top of a logic layer where memory controllers are implemented or a Physical Layer (PHY) that connects external links to the memory cells. All layers can communicate seamlessly with each other through the Through-Silicon Via (TSV), which is the technology that allows for vertical connection. They are logically split into up to 32 independent vaults or channels, which causes them to enable very high levels of data access parallelism. Due to their improved parallelism, 3D-stacked memories generally provide improved bandwidth. Table 2.1 shows maximum bandwidth rates for the most common memory devices. However, utilization of this bandwidth is often limited by the width of the connections between memory and host processor. By placing processing elements on the logic layer of a 3D-stacked memory, NDP designs bypass this limitation and are able to use the bandwidth these devices offer more efficiently and thus improve performance for many data-driven applications. A wealth of NDP proposals in the literature have targeted 3D-stacked memories (Alves et al., 2016; Tomé et al., 2018; Liu et al., 2018; Azarkhish et al., 2018; Schuiki et al., 2018; Thottethodi et al., 2018; Gao et al., 2015, 2017; Min et al., 2019; Oliveira et al., 2017; Cordeiro et al., 2021; Sim et al., 2018; Deng et al., 2018; Kwon et al., 2021; Lee et al., 2018; Zhu et al., 2013b,a; Nair et al., 2015; Ahmed et al., 2019; Kepe et al., 2019; Mirzadeh et al., 2015; Santos et al., 2017; Pugsley et al., 2014; Drumond et al., 2017, 2018; Gao and Kozyrakis, 2016; Boroumand et al., 2018).

Although not DRAM-based, the organization of non-volatile memories still follows the same principles in an effort to increase bandwidth. Memristors are a type of non-volatile memory commonly used in NDP research: their connection to a crossbar allows access to entire rows of memory cells at once, which is desirable for NDP. Thus, in theory, when considering this type of organization and access, the larger are the rows used in the memory layout, the larger the NDP capabilities (Chi et al., 2016; Shafiee et al., 2016; Cheng et al., 2017; Song et al., 2017;

Haj-Ali et al., 2018; Gupta et al., 2018; Imani et al., 2019; Drebes et al., 2020; Song et al., 2018; Huang et al., 2020; Angizi et al., 2019; Gupta et al., 2019; Sun et al., 2017; Jain et al., 2018; Xie et al., 2019)

Regardless of the underlying memory technology being used, NDP endeavours leverage memory bandwidth to extract performance. Thus, theoretically, the more bandwidth a memory can provide, the more execution time performance and energy efficiency a NDP architecture can extract from it. Table 2.1 compares the bandwidth achieved by common DRAM-based memories.

Table 2.1: DRAM architectures' performance comparison.

Memory Name	Maximum Bandwidth	Maximum Speed*	Energy Usage	JEDEC Compliant	Channels/ Vaults
DDR	3.2 GB/s	0.4 GT/s	257.13 pJ/b	Yes	2
DDR2	6.4 GB/s	0.8 GT/s	121.44 pJ/b	Yes	2
DDR3	14.9 GB/s	1.8 GT/s	64.70 pJ/b	Yes	3
DDR4	25.6 GB/s	3.2 GT/s	38.67 pJ/b	Yes	4
DDR5	41.6 GB/s	5.2 GT/s	N.A.	Yes	4
HMC	320 GB/s	2.5 GT/s	10.82 pJ/b	No	32
HBM	128 GB/s	1.2 GT/s	N.A.	Yes	8
HBM2	256 GB/s	2.0 GT/s	N.A.	Yes	8
HBM2e	410 GB/s	3.2 GT/s	N.A.	Yes	8
HBM3	819 GB/s	6.4 GT/s	N.A.	Yes	16

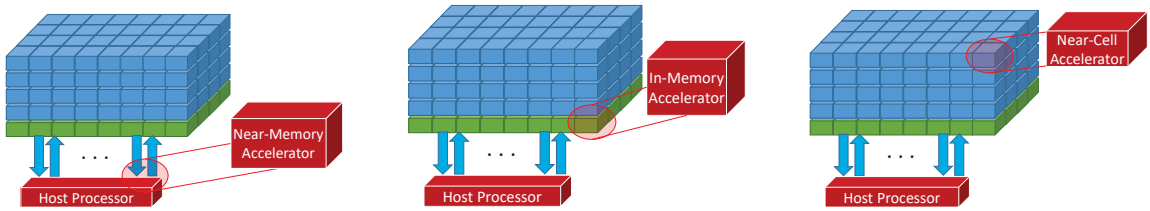
* Data rate/pin. ** From 2018.

2.2 DISTANCE FROM THE DATA

In general, near-data architectures look to leverage their advantageous position relative to the data storage to use the internal bandwidth of the memory device being used for data storage. Since these circuits and/or devices are placed very close to data storage elements of the system, they are often able to avoid the limitations of communicating with the memory through a narrow data bus, which gives them easier access to this available bandwidth. By doing so they also strive to reduce the costs of moving data between storage device and processing element, as data movement is the main source of inefficiency in modern systems (Zhang et al., 2013), thus improving performance regarding execution time and energy efficiency.

The choice of the location at which to implement NDP capabilities in a system has several consequences, such as how the processing elements access the data, how complex and energy efficient it can be, what sorts of operations it will be able to implement, etc. Considering distinct memory technologies and target applications, there are several different possible placements within a computer system for a near-data architecture. Figure 2.1 displays the three placements that are most common for near-data processing capabilities according to distance between processing element and stored data.

Near-cell accelerators implement logic within the memory cells that store the data, either by modifying their circuitry or the behavior of the memory device during data transferring between cells and row buffers (sense amplifiers), basically using analog signals. This is often done with DRAM-based memories (Li et al., 2017; Sim et al., 2018; Deng et al., 2018, 2019; Angizi et al., 2020; Huangfu et al., 2020; et al., 2017), but also with memristors, which are inherently capable of processing data (Chi et al., 2016; Shafiee et al., 2016; Cheng et al., 2017;



(a) Near-Memory Accelerators (b) In-Memory Accelerators (c) Near-Cell Accelerators

Figure 2.1: The most common types of NDP architectures. The blue blocks represent storage cells, green blocks represent logic layer (for 3D designs) or the sense amplifier (for 2D designs) and the arrows are the interconnection to outside the memory chip.

Song et al., 2017; Haj-Ali et al., 2018; Imani et al., 2019; Song et al., 2018; Huang et al., 2020; Angizi et al., 2019; Gupta et al., 2019; Sun et al., 2017; Jain et al., 2018; Xie et al., 2019). Figure 2.1(c) illustrates the placement of this class of accelerator relative to the data. Accelerators that apply this approach drastically reduce data movement, as processing is done as close as possible to the memory cells that store the data themselves. They can thus explore as much internal bandwidth as is possible, being almost optimally energy-efficient. The main drawback of this strategy is that, since it is implemented so close to the actual memory cells, it is limited to simple bit-wise logic operations (Gao et al., 2019; Seshadri et al., 2018), requiring additional hardware or modified storage layouts to support more complex operations (Eckert et al., 2018; Gao et al., 2019; Ali et al., 2020).

In-memory accelerators place processing logic within the memory device, but not directly on memory cells, as depicted on Figure 2.1(b). These devices commonly use 3D-stacked memories, implementing processing logic on the logic layer of the device. 3D-stacked memories are composed of several DRAM layers vertically connected through TSV and logically divided into up to 32 independent vaults, allowing for very high internal bandwidth. They also include a bottom layer that can implement simple logic, thus avoiding the issue of efficiently integrating logic and storage on the save device and enabling 3D-memories to perform processing tasks near-data, bypassing the need for off-chip data movement between memory and processor for such tasks.

Although some proposals use Single Data Rate (SDR) and DDR memories to achieve the same effect by placing logic alongside DRAM buffers, this is considered a very challenging task because logic- and storage-optimized technologies have opposing requirements, meaning integrating the two on the same circuit is likely to yield inefficient designs (Jacob et al., 2010). It is therefore more common for researchers to choose to use 3D-stacked memories for this type of NDP approach as such devices are able to integrate processing elements and storage circuitry but on separate layers of the device.

This type of approach efficiently leverages internal memory bandwidth by accessing data as closely as possible to the DRAM row buffers, thus avoiding any off-chip transferring of data and achieving high parallelism. When considering 3D-stacked memories, these proposals rely largely on the wider data buses of TSVs to move data between the memory and the processing elements (Wei et al., 2005; Farmahini-Farahani et al., 2014). Many in-memory accelerators implement full processing cores near the data (Azarkhish et al., 2018; Liu et al., 2018; Schuiki et al., 2018; Thottethodi et al., 2018; Gao et al., 2015; Cadambi et al., 2010; Lee et al., 2018; Ahn et al., 2015a; Nair et al., 2015; Pugsley et al., 2014; Alian et al., 2018; Devaux, 2019; Zhang et al., 2014), while others choose to employ only a set of functional units for SIMD-based processing (Alves et al., 2015b, 2016; Tomé et al., 2018; Oliveira et al., 2017; Cordeiro et al., 2021; Hsieh et al., 2016b; Ahmed et al., 2019; Kepe et al., 2019; Santos et al., 2017; Drumond

et al., 2017, 2018), or Application-Specific Integrated Circuits (ASICs) that can only accelerate applications pertaining to a specific domain (Liu et al., 2018; Azarkhish et al., 2018; Schuiki et al., 2018; Gao et al., 2017; Min et al., 2019; Kwon et al., 2021; Zhu et al., 2013b,a; Mirzadeh et al., 2015). The main drawbacks of this approach are that, since designs are confined to a logic layer, they are limited in terms of power consumption heat dissipation and available area, all of which are significant restrictions especially when a proposal wishes to implement complex structures near the data, such as a full processing core or a cache hierarchy. This is the approach we will be pursuing with the architecture we are proposing in this thesis.

Near-memory accelerators are implemented on a separate circuit than the memory device and placed either next to it (inside the same module) using an interposer or outside the memory module, connected to the memory through an off-chip interconnection (Hsieh et al., 2016b; Nai et al., 2017; Xi et al., 2015; Pugsley et al., 2014, 2015; Farmahini-Farahani et al., 2015; Drumond et al., 2017, 2018; Alian et al., 2018; Zhang et al., 2014; Kocberber et al., 2013; Hashemi et al., 2016; Awan et al., 2016). Figure 2.1(b) illustrates this placement. This approach is subject to fewer restrictions regarding area or energy consumption and is aimed mainly at reducing memory access latency. However, since it relies on off-chip communication, it may still suffer from higher latency and energy consumption.

2.3 PROCESSING MODEL

Although all NDP solutions aim to mitigate the memory wall and thus share the same objective, the most common approaches to it differ fundamentally (Loh et al., 2013). The decision on what kind of processing element to employ to achieve the actual near-data computation is one of the most consequential when designing a NDP architecture, as this choice will impact many details regarding integration of the near-data accelerator with the rest of the system.

2.3.1 Full-Stack Processor

One common approach to NDP is to integrate a full processing core to the memory device. Doing so partially avoids two large issues NDP adoption faces: (i) how accessible devices are to program and (ii) how data coherence is maintained. However, this approach is heavily limited by power and area constraints in the memory chips (Lima et al., 2018). Many solutions found in the literature propose this approach nonetheless (Liu et al., 2018; Ahn et al., 2015a; Drumond et al., 2017; Devaux, 2019; Zhang et al., 2014; Boroumand et al., 2018), adding full general-purpose cores to the logic layer of 3D-stacked memories.

Although traditional solutions function when considering a near-data core in isolation, challenges such as programming model and cache coherence still apply. With another full core in the architecture, the system must now be able to handle data coherence with an off-chip cache hierarchy, determine how programs being executed in the near-data core and programs running in the host processor can coexist, among other issues. Moreover, power and area requirements severely limit feasibility (Santos et al., 2019b,a, 2021a).

2.3.2 Functional Units

One of the very first attempts at NDP employed a different strategy: implementing Functional Units (FUs) within memory modules, thus having easy access to the internal bandwidth of a memory. Back in the 1990s, Computational-RAM (C-RAM) (Elliott et al., 1999) proposed adding functional units alongside DRAM row buffers, allowing for bitwise computation. The idea was considered impractical at the time as it had significant energy consumption and complexity

requirements: (i) it added a large number of functional units to the design (one vector of FUs per memory array), (ii) it required the operating system to maintain application-specific data mappings and (iii) it assumed an efficient integration of logic and storage elements on the same die.

With the development of TSV technology and subsequent 3D-stacking integration, the approach started to see a resurgence in the 2010s with many NDP proposals choosing to employ it. The FU-based approach benefits from being much simpler than integrating a full-stack processor and offering good execution time performance while being very energy efficient. However, it suffers from all aforementioned integration challenges and requires specific solutions regarding programming models, cache coherence and support for virtual addressing (Santos et al., 2019b,a, 2021a). These issues are so severe that many of the existing research proposals bypass or neglect some of them completely by, for instance, reserving exclusive memory space for NDP operations, changing logical memory types or simply not providing virtual memory or cache coherence solutions (Drumond et al., 2017; Devaux, 2019). Although it could be argued that cache coherence is not necessarily to be solved by hardware and that such checks could be left to the programmer, most related work does not discuss this. Nonetheless, it is a very popular approach in NDP research, with many proposals using FU-based devices to exploit memory bandwidth (Alves et al., 2015b, 2016; Tomé et al., 2018; Oliveira et al., 2017; Hsieh et al., 2016b; Alves et al., 2015a; Ahmed et al., 2019; Kepe et al., 2019; Xi et al., 2015; Santos et al., 2017; Drumond et al., 2017, 2018). A current challenge and significant limitation that all these ideas face is that all of them require extensively modifying host processors to support their NDP logic, as no existing systems offer native support.

2.3.3 Computing in Memory Cells

Another approach is to exploit the analog circuitry of memory devices to process data, thus effectively considering the memory a processing device itself. This is achieved by modifying memory circuits to allow for processing while transferring data between the memory cells and the row buffers (i.e. sense amplifiers) of the device. In DRAM-based designs, for instance, this can be accomplished by sharing capacitor charges (Deng et al., 2018; et al., 2017; Gao et al., 2019; Hajinazar et al., 2021; Seshadri et al., 2018). Similarly, newer technologies such as Resistive random-access memory (ReRAM) and Spin-Transfer Torque Random-Access Memory (STT-RAM) exploit Kirchoff's Law and use electrical resistance to compute on store data (Eckert et al., 2018; Aga et al., 2017; Li et al., 2017; Chi et al., 2016; Shafiee et al., 2016; Song et al., 2017; Drebes et al., 2020; Song et al., 2018; Xin et al., 2020; Jain et al., 2018; Xie et al., 2019).

This approach faces many of the same issues as FU-based solutions in that is also requires significant modifications to hosts. It is also limited strictly to bitwise operations, which poses a significant challenge to implementing complex instructions. Since most of these proposals rely on technology that is not yet commercially available, availability is also limited for solutions regarding cache coherence, programming model and virtual addressing that consider this type of approach. (Gao et al., 2019; Hajinazar et al., 2021; Santos et al., 2019b).

2.3.4 FPGA/CGRA

A fourth approach to NDP is to place Field-Programmable Gate Array (FPGA) or Coarse-Grain Reconfigurable Array (CGRA) in the memory (Farmahini-Farahani et al., 2015; Gao and Kozyrakis, 2016; Kara et al., 2017; Singh et al., 2021). Although the high area and power requirements may be difficult to reconcile with the limitations of NDP designs, they can allow

for near-optimal hardware-software integration as they provide the possibility of on-demand implementation of functional units.

Most issues faced so far also apply to this approach: they require cache coherence and virtual memory solutions. They also create the additional requirement of *bitstream* emission, which describes the hardware configuration they must implement and for which a solution must also be provided.

2.3.5 Specialized Accelerators

Lastly, specialized Accelerators or ASIC, are designed to provide acceleration to a specific class of application, such as neural networks or network intrusion detection systems. Many specialized NDP accelerators can be found in the literature and, although some of them may use similar approaches to ones already discussed, they are usually designed in a way that avoids issues and requirements that apply to general-purpose proposals (Ando et al., 2017; Liu et al., 2018; Azarkhish et al., 2018; Schuiki et al., 2018; Gao et al., 2017; Min et al., 2019; Kwon et al., 2021; Nai et al., 2017; Cali et al., 2020; Zhu et al., 2013b,a; Mirzadeh et al., 2015; Pugsley et al., 2015). The application-specific nature of these devices will usually allow for workarounds to maintain data coherence and support virtual addressing without requiring modifications to the host processor.

2.4 PROGRAMMING MODEL

Every NDP architecture must provide a way for the host processor of the system to offload tasks and/or instructions to be executed near data. Many different strategies can be used to achieve this and in this section we discuss such strategies and how they are influenced by other aspects of the design of a NDP device.

2.4.1 Host Triggers Instructions

FU-based and in-memory approaches to NDP exploit features that are inherent to memory devices, with little room for complex structures. While simple elements such as functional units can feasibly be implemented, complex mechanisms such as instruction fetching and decoding cannot be reasonably expected from such types of devices. It is therefore common for proposals that fall under those categories to rely on the host processor of the system to trigger instructions to the NDP device.

By adopting this strategy, tasks like instruction fetching and decoding are kept in the host processor. Since instructions are already found in the L1 cache most of the time and decoding is a complex operation, keeping such functions in the processor tends to be much more efficient than moving them to the memory. Naturally, this requires the processor to support the NDP Instruction Set Architecture (ISA), one of the aforementioned necessary host modifications. Moreover, this also means that binary code for applications that wish to utilize NDP capabilities will feature a mix of both traditional instructions from the host ISA and accelerator-specific instructions.

There are two main strategies to achieving such an effect, one of which is **manual code generation**, when code is written by the user, possibly with an Intrinsics library (Cordeiro et al., 2017). This approach assumes a compiler that is NDP-aware in that its generated binary will include the appropriate NDP instructions in place of the corresponding Intrinsics functions calls in the code. Code can be finely controlled by the programmer. NDP proposals that use this approach for programming assume the host processor will stay in charge of fetching and decoding instructions that will be then offloaded for near-data execution. Other than a modification

to the host processor ISA to support such instructions, this also requires the behavior of the cache hierarchy and Memory Management Unit (MMU) to be able to react to such additional instructions to maintain cache coherence.

A second strategy is **automatic code generation**, which relies on compiler support to generate NDP code automatically. This strategy assumes a compiler that is able to optimize code for NDP execution on a specific architecture, identifying opportunities for NDP offloading by itself. The compiler must be aware of all ISA options supported by the system and how they interact with one another in order to provide appropriate instruction sequences. An adequate NDP-aware compiler can, for instance, identify possible data coherence conflicts and either adjust or re-order instructions to alleviate coherence issues at compilation time.

One such compiler is Compiler-Assisted InstRuction-level Offloading (CAIRO) (Hadidi et al., 2017), which supports the native instructions in the Hybrid Memory Cube (HMC) ISA (Hybrid Memory Cube Consortium, 2014). CAIRO decides whether to include HMC instructions in the binary code by analyzing a number of information regarding the profile of an application. Such decisions are thus made with the aid of a cache profiling tool that records traces of past executions and provides information such as cache miss rate and bandwidth savings. Applications are compiled, profiled and analyzed by CAIRO, which then adds HMC to the binary code if deemed advantageous. CAIRO reportedly improves execution time performance significantly for selec applications but is limited to the atomic HMC instructions and does not consider SIMD instructions that may be supported by the processor.

Processing-In-Memory cOmpiler (PRIMO) (Ahmed et al., 2019; Santos et al., 2019a) is another compiler with full NDP support, with both offloading and optimization features. All NDP offloading and SIMD are considered at compile time as it is designed to take advantage of all hardware capabilities without programmer input with pragmas or directives.

2.4.2 Function or Binary Migration

For proposals that implement full processing cores near the memory, we expect such cores to handle traditional programming models (Liu et al., 2018; Ahn et al., 2015a; Drumond et al., 2017; Devaux, 2019; Zhang et al., 2014; Boroumand et al., 2018). These designs will usually be programmed with annotated code, relying on libraries to handle task offloading and communication with the host processor.

For instance, the Asynchronous Memory Compiler (AMC) (Nair et al., 2015) takes a directive-based approach to programming for NDP-enabled systems. It takes code written for OpenMP and analyses it to determine which portions of it can be offloaded to be executed by the AMC. Authors report an improvement of up to 70% better utilization of hardware resources when running specific scientific applications, such as matrix multiplication.

In a similar vein, Transparent Offloading and Mapping (TOM) (Hsieh et al., 2016a) takes a compiler-based approach to instruction offloading considering a Graphics Processing Unit (GPU)-based system with multiple 3D-stacked memories. It observes runtime conditions of the systems to determine when instructions blocks can be offloaded for execution on the 3D-stacked memories if doing so would yield enough memory savings to justify this decision. This solution relies on Compute Unified Device Architecture (CUDA) annotated code.

Another strategy that can be found in the literature is Bandwidth-Critical Data Analysis (BCDA) (Khaldi and Chapman, 2016), which presents itself as an extra step during the compiling stage in which the compiler identifies opportunities to transform regular memory allocation into HBM-specific allocation commands when beneficial. Although mostly automatic, the process still requires some attention from the programmer, as it relies on special functions. The authors

of BCDA report an improvement in performance of up to 2.33× in execution time with their approach when running a Conjugate Gradient benchmark.

The HBM-PIM (Kwon et al., 2021) also relies on a function offloading model, using annotated code and requiring a specific Application Programming Interface (API). The host is tasked with either transmitting a code snippet describing the operation and the memory addresses over which the code must be computed.

2.4.3 Hardware Driven

Yet another approach relies on in-flight instruction offloading, by monitoring hardware metrics and behavior to determine when instructions or tasks must be offloaded for NDP execution. For example, one proposal uses metrics monitoring and an ISA extension to perform near-data acceleration at the memory controller by executing operations over data as soon as it arrives from the main memory (Hashemi et al., 2016). The authors report a 20% reduction in latency from dependent cache misses by using their approach.

2.5 3D-STACKED MEMORY DEVICES

Since the early 2010s, two 3D-stacked memories have become commercially available, HMC and High Bandwidth Memory (HBM). The Hybrid Memory Cube was first announced in 2011 by Micron, followed by a full specification and release in 2012 (Hybrid Memory Cube Consortium, 2012). It was described as "a single package containing multiple memory die and one logic die, all stacked together using Through-Silicon Via (TSV) technology" (Hybrid Memory Cube Consortium, 2012). It was logically split into 16 independent vaults, each with its own exclusive memory controller and banks, and supported memory requests of up to 128 bytes of data. The device was revised in 2014, seeing an increase in maximum number of logical vaults (from 16 to 32) and support for 256-byte requests (Hybrid Memory Cube Consortium, 2014). Figure 2.2 shows a block diagram of the device (Hybrid Memory Cube Consortium, 2012). As shown, the Hybrid Memory Cube connects to the system through a crossbar switch.

In August 2018, Micron announced it was moving away from the HMC and focusing on other high-performance memory technologies such as GDDR6 and HBM, following a decision by the Joint Electron Device Engineering Council (JEDEC) to make HBM an industry standard. The HBM was released in 2013 by AMD and Hynix, becoming the JEDEC standard by October of the same year. It started being produced *en masse* in 2015. The first specification (Association, 2013) described it as a 3D-stacked die composed of 8 independent vaults with 8 or 16 banks per independent vault. Its first major revisions were the HBM2 and HBM2E, in November 2018, when the maximum number of banks per vault was increased to 32 while row buffer size was reduced from 2 KB to 1 KB. In January 2022, the HBM3 was accepted as a JEDEC standard, further modifying the device to allow up to 16 independent vaults, each containing up to 64 banks (Association, 2022). Figure 2.3 shows a diagram of the High Bandwidth Memory, lifted from the manufacturer's website. Unlike the Hybrid Memory Cube, the HBM assumes a interposer-based connection.

2.6 SUMMARY

In this chapter we discussed many aspects that pertain to the design of NDP architectures. As we have seen, the placement of the near-data solution relative to the stored data greatly informs several aspects the design of an architecture, impacting what performance can be expected of

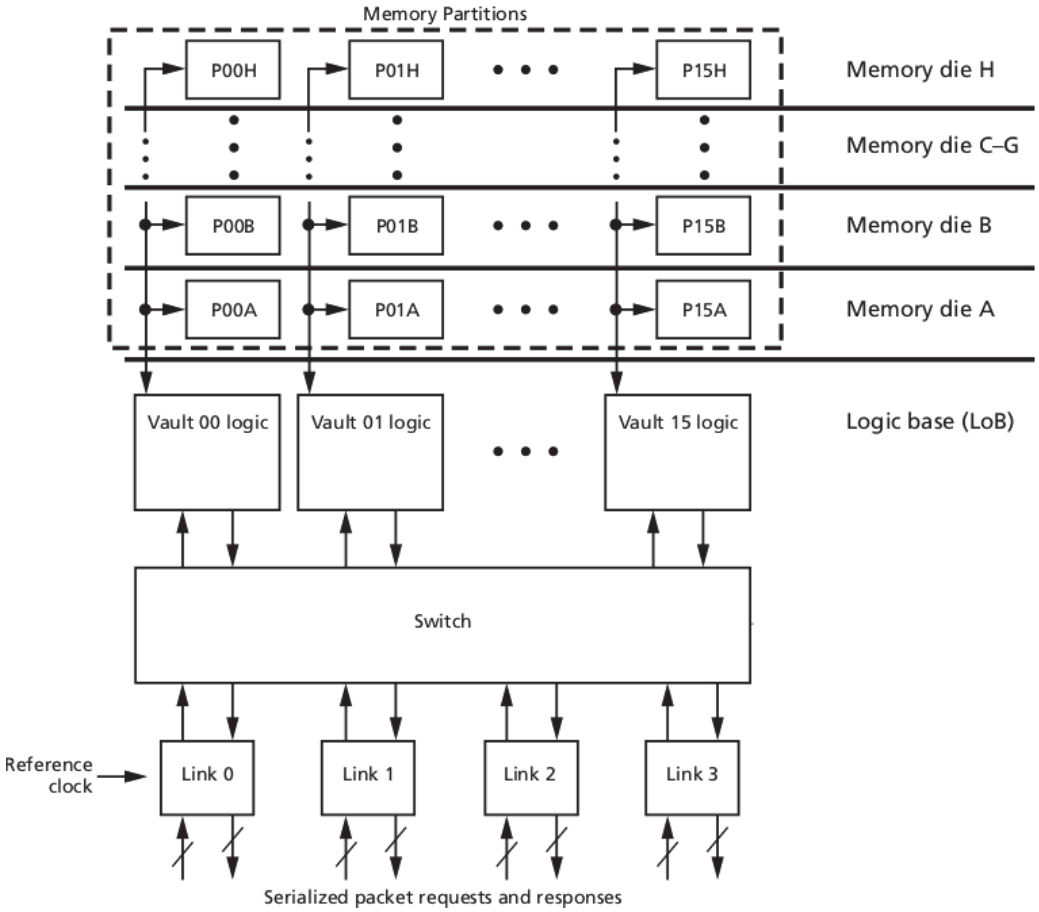


Figure 2.2: Hybrid Memory Cube Block Diagram Example Implementation (Hybrid Memory Cube Consortium, 2012).

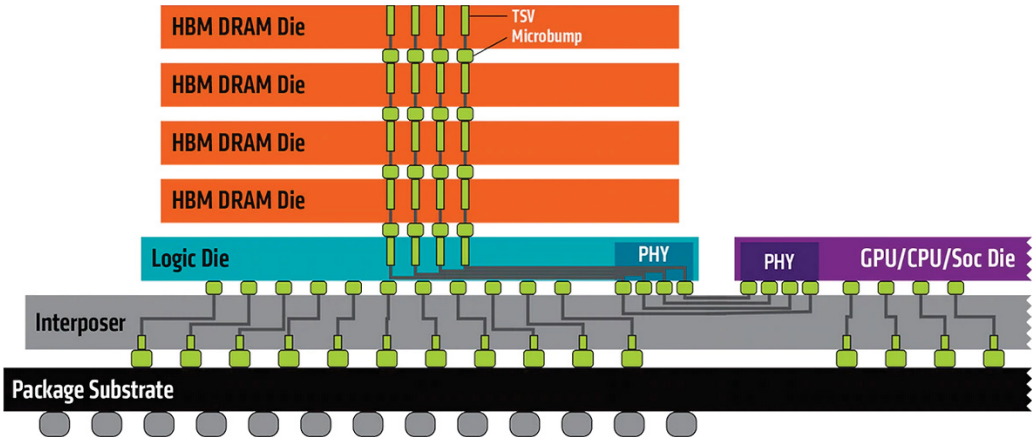


Figure 2.3: High Bandwidth Memory scheme. (AMD, 2015).

a NDP device, especially regarding data bandwidth. We also discussed how different types of memory influence its utilization in the context of a computer system, which informs what kind of data it stores, what it is used for and what benefits an NDP can derive from them. Next, we spoke of the several ways in which the actual processing of operations can be implemented near the data and of the benefits and limitations of each approach. Our discussion of NDP programming models went over different strategies of communication between host processor and NDP, which

directly informs how such accelerators can be programmed. Lastly, we discussed the most common 3D-stacked memory designs.

This discussion informed many aspects of our proposal. VIMA assumes its placement on the logic layer of a 3D-stacked memory, thus positioning itself as a DRAM-based in-memory accelerator. This placement choice avoids both the limitations in possible types of operations faced by near-cell designs and the higher latency and energy consumption of near-memory accelerators. Although all NDP processing models must address integration issues, we chose to use an FU-based model. We find that this choice of processing element combines low energy consumption, area and complexity requirements with high processing power and will discuss it further in Section 4.10. Lastly, as a consequence of our FU-based approach to processing, our programming model is based on instructions that are added to the processor ISA and offloaded to VIMA after the fetch and decode stages of the processor pipeline. Further discussion on this can be found on Sections 4.1 and 4.2. The work presented on this chapter has been published on the Journal of Integrated Circuits and Systems (Santos et al., 2021b).

3 RELATED WORK

In this chapter we discuss some existing work in NDP research in order to grasp the state-of-the-art in the area. NDP can be used to run any number of algorithms, but it is most suited to those that present two specific characteristics: (i) a **data streaming behavior**, causing a near constant flow of data either to or from the memory; and/or (ii) a possibility for **coalescent data access**, meaning the application would allow for a large number of contiguous data elements to be processed at once, which would lead to increased opportunities for exploitation of increased available data bandwidth.

Figure 3.1 (Santos et al., 2021b) displays results of an experiment designed to illustrate this point. The experiment models a multi-core processor with a 16 MB Last-Level Cache (LLC) and executes a simple application that consists in comparing elements in an array of integers. Results in the illustration regard how an NDP architecture performs against such a traditional system, with values greater than 1 meaning it achieved superior execution time performance. The application iterates over the array several times to illustrate how the size of the cache memory in the traditional system affects the final results. Moreover, the number of threads in the baseline also varies to simulate increased pressure being applied to the main memory.

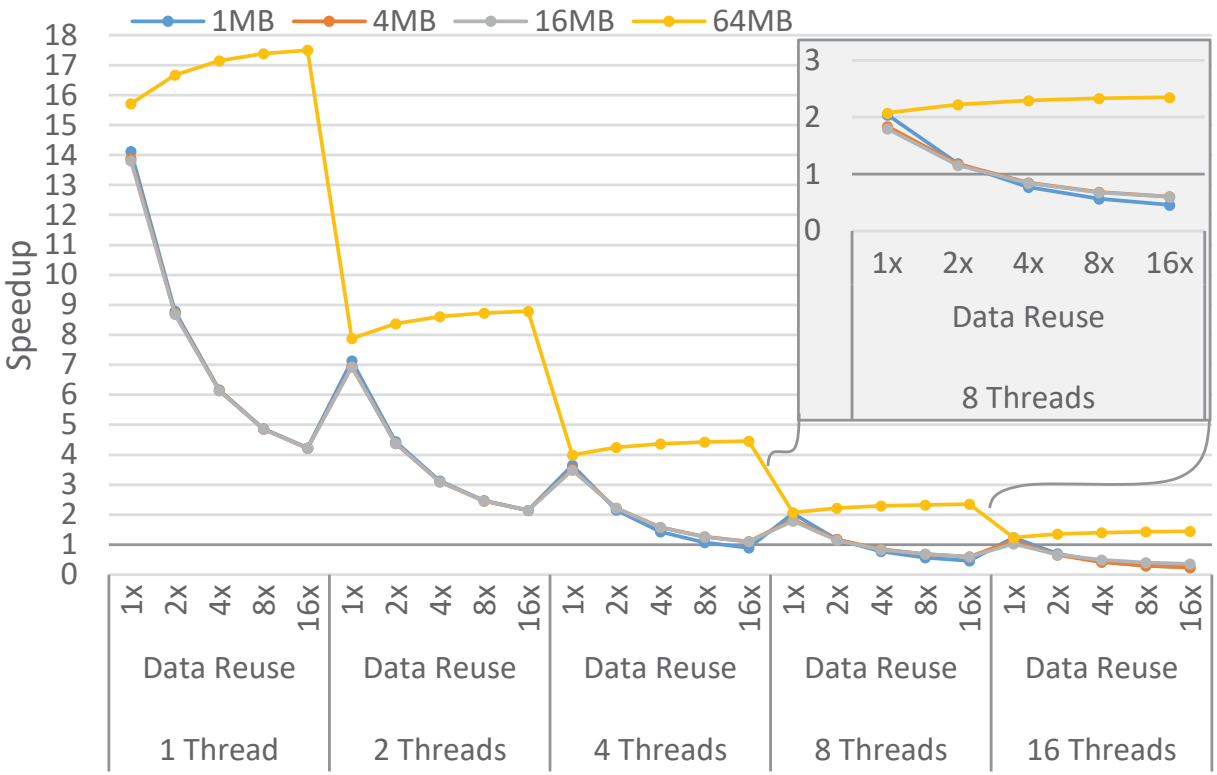


Figure 3.1: NDP performance compared to traditional x86.

As can be seen in Figure 3.1, applications for which the memory footprint is smaller than the size of the LLC or that present clear data reuse behaviors benefit greatly from the cache hierarchy in the traditional baseline system. Meanwhile, whenever memory footprint is larger than LLC or there are no data reuse opportunities for the cache memory, migrating the application to NDP execution becomes advantageous. This insight gives us some guidance as to what types of applications may be most suitable for NDP, and proposals in the literature have mainly sought

to accelerate application domains with such patterns. In the next sections, we discuss several NDP works found in the literature, according the application domains they claim to support.

3.1 NEURAL NETWORKS

Neural networks are used to recognize patterns in data and classify data points into categories, relying on analysis of vast amounts of data to do so. Algorithms are commonly split into (i) a training phase, in which a dataset is used to train model parameters; that are then used in (ii) the classification phase, where a much larger dataset is then classified according to the parameters and categories that were set in the previous phase. Neural networks are composed of layers of neurons, each of which has a set of activation values and connections weights that inform how layers relate to each other. The most common operation in neural network algorithms is the product of these values, which happens repeatedly from input to output layer in a behavior known as forward-propagation. The opposite operation, backward-propagation, happens when the weight of connections between layers is updates from output layer to input layer. Both operations happen several times during the training phase of the network and, depending on the size of the data set being used for training, these algorithms can present a data streaming behavior. As a result, it is common for traditional systems to offer long execution times and poor energy efficiency when running such algorithms.

Many proposals that aim to accelerate neural networks with NDP have focused on SRAM, mainly addressing binary and ternary networks, as SRAM usually has lower storage capacity. For instance, Neural Cache (Eckert et al., 2018) adds processing capabilities to bit-line peripherals in the LLC, thus providing bit-serial FUs that can be used for neural network computation. Neural Cache has been extended or improved upon several times. One such extension by Wang et al. (2019) adds many optimization features to the original design, which uses sparsity-awareness and Neural Network (NN) redundancy to improve performance while also applying more efficient computing techniques to binary and ternary neural networks. (Yin et al., 2019) (2019) takes the same approach as Neural Cache but modifies it to improve scalability. To do so, it enables activation of multiple SRAM rows with XNOR-Accumulate operations, hides in-memory reprogramming latencies with double buffers and adds peripheral logic to allow for multi-bit activation.

Other researchers apply a different idea. For instance, Ando et al. (2017) and Takamaeda-Yamazaki et al. (2017) split SRAM memories into regions in which distinct elements of binary and ternary NNs are stored, which allows for more efficient processing. BFree (Ramanathan et al., 2020) is a LUT-based NDP proposal that places processing elements on SRAM subarrays and allows for adaptable NN layouts and precision. The work of Long et al. (2020) uses well-known convolutional NNs such as LeNet, AlexNet, VGGNet and ResNet to demonstrate how their optimization further optimizes SRAM-based NDP architectures for this application domain. Lastly, Compute Cache (Aga et al., 2017) modifies cache memories by adding vector operation support to them, allowing for computation of large data operands.

While SRAM-based NDP systems perform adequately, DRAM-based proposals have largely been more popular for large-scale neural network near-data acceleration. Many of these works implement full-cores close to the memory, as is the case with the Convolutional Neural Network (CNN) accelerators proposed by Liu et al. (2018), Azarkhish et al. (2018), Schuiki et al. (2018), and Thottethodi et al. (2018). The work of Liu et al. (2018), for instance, implements ARM-based cores on the logic layer of a 3D-stacked memory. The cores are fully programmable and a handle a number of function that can be offloaded to them by the host processor. Gao et al. (2015) also employ ARM cores and use a number of mechanisms such as a Translation

Look-aside Buffer (TLB) and virtual memory scheme to enable inter-vault communication in the memory through a vault router. In similar efforts, Azarkhish et al. (2018) and Schuiki et al. (2018) implement RISC-V cores modules equipped with cache memories and Direct Memory Access (DMA) that control smaller processing elements near-data. This work features a set of multiple instances of the HMC that communicate with each other. Thottethodi et al. (2018) also accelerate neural networks by implementing full processing cores on the memory layer of a 3D-stacked memory, each with its own pipeline register files and prefetching capabilities. Meanwhile, Gao et al. (2017) implement simple cores composed of Arithmetic Logic Units (ALUs) and a register file. Each vault in a 3D-stacker memory has its own core and they all share a dedicated network and global buffer to enable parallel processing. The work of Min et al. (2019) focuses on communication and implements a Network-on-Chip (NoC) to enable communication between the vaults of a 3D-stacked memory and extract performance through higher parallelism. It is used to accelerate a Deep Neural Network (DNN) by splitting and scheduling computation across the various vaults of the memory device.

With a FU-based approach, Oliveira et al. (2017) implement NIM, which places a processing module near the DRAM. Each module features a set of FUs, a register bank and a sequencer and is attached to an individual vault of a 3D-stacked memory. It is used to simulate biologically meaningful NNs.

Works such as the proposals of Li et al. (2017), Deng et al. (2019), Sim et al. (2018), Deng et al. 2018), and Cadambi et al. (2010) take yet another approach and add logic ports and bitwise operations to conventional DRAM, which are then used to accelerate convolutional and binary neural networks near the data. While Sim et al. (2018) and Cadambi et al. (2010) use these added circuits to partially calculate NN layer propagation values and thus speedup computation, Li et al. (2017), Deng et al. (2018), and Deng et al. (2019) implement reconfigurable circuits inside the memory.

With the advent of efficient implementation of the memristor, many researchers have proposed ReRAM-based approaches to near-data acceleration of neural networks. Because of the crossbar array organization of memristors, a dot product operation can be implemented very efficiently with this type of memory. PRIME (Chi et al., 2016) is one such proposal, in which a ReRAM main memory has subset of memory arrays that can be configured to be used exclusively for neural network inference acceleration. The TIME architecture, proposed by Cheng et al. (2017), improved upon PRIME by adding circuitry to the design to support weight updates and optimizations to mitigate the impact of writing operations to the memristors. ISAAC (Shafiee et al., 2016) is another such proposal. It implements a efficient dot product operation that is then used to accelerate DNN inference. PipeLayer (Song et al., 2017) extends ISAAC to support the training phase of a DNN. It optimizes the original design to simplify its pipeline, reducing the impact of bubbles and eliminating signal conversions. Haj-Ali et al. (2018) take the existing MAGIC architecture (Kvatinsky et al., 2014) and extend it to add support to complex operations, thus enabling image-processing tasks. FELIX (Gupta et al., 2018) is an in-cell 1-cycle logic implementation aimed at enabling complex operations used to classify images using neural networks. FloatPIM (Imani et al., 2019) is a mechanism that uses NOR memristor applications to implement floating-point representations and operations. This directly impacts the performance of PipeLayer (Song et al., 2017) and ISAAC (Shafiee et al., 2016) by improving execution time and inference accuracy. FIMDRAM (Kwon et al., 2021) adds SIMD processing capabilities to the HBM in an effort to take advantage of the bank-level parallelism present in the device and achieves a 4× speedup in comparison to an off-chip device according to the authors. SSAM (Lee et al., 2018) is an similarity search accelerator for kNN, which reportedly outperforms both GPU- and FPGA-based alternatives regarding throughput and energy efficiency. PIMS (Li et al., 2019)

is also built into the HMC and adds an accelerator per vault. Instead of adding new processing elements to the device, this proposal uses the device's processing capabilities. It simply adds new logic that implements a specific request/response dispatcher. The architecture aims at improving the processing of 3D stencil algorithms and reports a 72% reduction in redundant memory accesses for workloads larger than the host's LLC. Lastly, TC-CIM (Drebes et al., 2020) is an end-to-end compilation framework aimed at enabling high throughput for NDP machine learning inference accelerators. It uses tensor comprehensions and loop tactics to exploit NDP capabilities automatically, thus providing an interface for NN acceleration using different memory technologies.

3.2 GRAPH TRAVERSING / POINTER CHASING

Graphs are data structures that store data elements and also represent relationships between data points. The main issues faced by architectures when dealing with graph processing applications is that they will usually have a random behavior regarding data access, which leads to poor locality of reference. This severely hinders cache memory performance and causes workload imbalances when considering multiple processing units because depending on the behavior of the traversing algorithm being used, applications that deal with graphs can have very irregular memory access patterns, which degrades bandwidth usage between the memory and the CPU and causes inefficiencies in cache memory use. Such applications often require accessing data to determine the address of the next data element that shall be accessed, resulting in a situation known as pointer chasing. Also, since graphs are intrinsically irregular, graph processing applications are likely to suffer from unbalanced workloads and race conditions when running in a multithreaded environment.

Many authors have proposed using NDP to accelerate pointer chasing applications, most of them using DRAM technology. Tesseract (Ahn et al., 2015a), for instance, adds one full in-order core to the logic layer per vault of a 3D-stacked memory device. Each core accesses only its respective vault and a memory region that is marked non-cacheable, thus avoiding any possible coherence issues. The authors evaluate their performance using real-world graphs and report results 10× faster than the considered baseline while reducing energy consumption by 87%. Each memory vault has its own ARM processor, 512 in total, and they are all used to exploit the internal bandwidth of the 3D-stacked memory chips. IMPICA (Hsieh et al., 2016b) is another mechanism used for pointer chasing applications, aimed specifically at improving the performance of linked lists and B-trees by avoiding cache pollution when sparse memory access patterns are detected. It also used 3D-stacked memories, implementing several different engines in memory vaults. GraphPIM (Nai et al., 2017) is a full-stack solution for near-data graph traversing applications. In an effort to improve performance of graph computing frameworks, which usually perform poorly on modern computers due to their irregular memory access patterns and repetitious atomic operations, this work leverages the near-data atomic instructions offered by Micron's HMC (Hybrid Memory Cube Consortium, 2013). GraphPIM requires implementing an offloading unit in the processor and defining a non-cacheable region in the virtual memory space. This memory region is used to store the data most responsible for onerous atomic instructions: operations detected to be accessing an address within the bounds of the region are automatically offloaded to be executed near data in the HMC, avoiding costly cache coherence operations. Since this is done through setting a automatically detectable non-cacheable memory area, the solution does not impact programmability, as no code needs to be rewritten by final users. The authors report a speedup of 2.4× across a wide range of graph computing benchmarks while consuming 37% less energy on average. Hong et al. (2016) propose another mechanism that

uses NDP to accelerate linked lists. They use the HMC and their approach is to batch operations according to the location of the data to be processed in the memory. This approach argues that simply offloading tasks to be processed near the data may be disadvantageous if data locality is poor. By batching operations, they attempt to better utilize the internal bandwidth of the memory.

Finally, proposals such as GraphR (Song et al., 2018) and Hetraph (Huang et al., 2020) are based on ReRAM. GraphR uses a streaming-apply model and functions as an out-of-core pre-processing tool for graph processing while Hetraph is a hybrid architecture that uses ReRAM, traditional logic and software functions to fully exploit NDP capabilities.

3.3 GENOME SEQUENCING/PATTERN MATCHING

Widely used in bioinformatics, genome sequencing involves several tasks related to identifying DNA sequences in samples, including counting, alignment and sequence assembly. However, these tasks are non-trivial due to the size of the datasets involved and the characteristics of the application. For example, a single DNA sample generates tens of millions of sequences that must then be mapped to known datasets with billions of sequences. Thus, genome sequencing tasks suffer from memory-wall bottlenecks due to the massive amount of data access required. The same issue occurs with other big-data applications that similarly rely on pattern matching tasks, such as network security and data mining, increasing the interest for NDP to improve this application domain.

Two proposals use SRAM technology to execute efficient matching algorithms. The work of Sadredini et al. (2020) discusses how existing NDP pattern matching accelerators fail to use resources to their fullest extent. The authors propose a general-purpose pattern matching architecture called Impala that implements efficient multi-stride near-data processing automata by addressing these issues. GenASM (Cali et al., 2020), is a framework for approximate string matching designed for genome sequence analysis and used to accelerate various steps in the sequencing process. We also found two related works that explore DRAM technology. Angizi et al. (2019) propose AlignS (Angizi et al., 2019), which uses Spin-Orbit Torque Magnetoresistive Random-Access Memory (SOT-MRAM) and an assembler (Angizi et al., 2020) using DRAM-based NDP accelerators to aid DNA sequence alignment and assembly, respectively. Huangfu et al. (2020) created NEST, a NDP architecture that accelerates k-mer counting, which is another important task in DNA sequencing processes. RAPID (Gupta et al., 2019), on the other hand, is an memristor-based architecture that also supports DNA alignment tasks considering a parallel version of the state-of-the-art algorithm. Lastly, GenStore (Ghiasi et al., 2022) places accelerators inside a NAND flash-based Solid-State Drive (SSD) to achieve in-storage processing of genome sequence analysis tasks, reporting improvements in execution time between 2× and 19× against the state-of-the-art depending on patterns in the data.

3.4 COMPUTATIONAL FLUID DYNAMICS

Computational Fluid Dynamics (CFD) are used by applications in many scientific domains. CFD algorithms often apply similar kernels, such as matrix convolutions and multiplications, thus being prime candidates for vectorization with SIMD-based processing.

Zhu et al. (2013b) place existing custom cores (Zhu et al., 2013a) on the logic layer of a HMC to aid in sparse matrix multiplication operations, which are commonly used by graph applications and to compute the Fast Fourier Transform. Also based on the HMC, the AMC architecture (Nair et al., 2015) places vector registers on the logic layer of the device and implement a number of mechanisms near the data. It includes a vector instruction set, predicated

execution, virtual addressing and gather-scatter operations that are applied directly to the store data. Memory Vector eXtension (MVX) (Alves et al., 2015b,a) implements a set of FUs inside the DRAM to perform a number of operations near-data, allowing for data to be accessed directly on the DRAM row buffers and processed by SIMD functional units.

3.5 DATABASE APPLICATIONS

Analytical database applications are prime examples of workloads that can benefit from NDP offloading. They commonly move vast amounts of data from the memory to the processor in large workloads that attempt to identify relationships and patterns in data. Such applications will often pollute cache hierarchies due to their data streaming behavior and, consequently, low locality of reference. Analytical database queries usually consist of a chain of database operations ordered in a way that extracts a specific result from the stored data. The most common operations are selection, projection, join and aggregation, all of which account for a combined 90% of the execution time and memory usage of the TPC-H set of analytical database benchmarks (Kepe et al., 2019). Such operators are commonly targeted by NDP proposals that aim to accelerate database applications.

One such proposal is brought forward by Kim et al. (2011), which modified the memory controllers of an SSD storage device to enable it to execute the selection operatior. et al. (2017) propose Ambit, which implements in-cell bit-wise operations on DRAM memories requiring only minor changes. HIPE (Tomé et al., 2018) places processing elements on the logic layer of a 3D-stacked memory to implement near-data filters using predicated instructions. RVU (Santos et al., 2017) enables reconfigurable in-memory vector processing by placing processing units on the memory vaults of a HMC device. JAFAR (Xi et al., 2015) also focuses on the selection operator, connecting a dedicated off-chip circuit to the memory I/O which works as a near-data filtering device. Sun et al. (2017) leverage memristor crossbar access patterns to implement several operators with an in-cell NDP strategy using ReRAM. Lastly, Kepe et al. (2019) analyze the performance of database operators being executed on a NDP-enabled 3D-stacked memory and a traditional x86 system to investigate situations in which each architecture is most advantageous for operations to use.

3.6 MAPREDUCE

The MapReduce model is widely used to automatically distribute the process of extracting information from large datasets among several parallel threads. It follows the Map and Reduce primitives: the map phase is used to classify data, producing an intermediate result that is then distributed among processing nodes, at which point the reduce function is used to group elements with similar features. The process may require several MapReduce instances working in parallel and usually requires moving large amounts of data between the memory and the processing cores.

NDCores (Pugsley et al., 2014) is an NDP architecture that supports MapReduce workloads and add 512 complete cores to the logic layer of a HMC device. By leveraging the superior parallelism 3D-stacked memory devices offer, they aim to speed up the execution of MapReduce applications, which are naturally parallel. They consider a chain of 3D-stacked memory devices and report reductions of up to 15× in execution time and 18× in energy consumption when comparing their NDP proposal to an optimized traditional MapReduce execution. Farmahini-Farahani et al. (2015) proposes attaching FU-based accelerators to commodity DRAM chips through TSV to improve performance by leveraging the high parallelism of MapReduce memory accesses, reportedly outperforming a traditional x86 architecture by

67%. The Mondrian data engine (Drumond et al., 2017, 2018) takes a hybrid approach that combines software and hardware using 3D-stacked memory-based NDP to implement data analytics operators that are used during the partitioning and shuffling steps of MapReduce. Meanwhile, Memory Channel Network (MCN) (Alian et al., 2018) proposes using NDP to directly support MapReduce frameworks Hadoop and Spark from the perspective of a server cluster.

3.7 MULTIPLE DOMAINS

Lastly, several proposals attempt to use NDP to accelerate applications in multiple domains, including those already discussed in this chapter.

The UPMEM PIM architecture (Gómez-Luna et al., 2021) integrates general-purpose in-order cores with DRAM chips, coexisting with a regular main memory. Each NDP chip has its own 64 MB DRAM bank, an instruction memory and a small scratchpad memory. Communication with the host consists in copying data to and from the main memory, and near-data processing units do not communicate directly, meaning the architecture is geared towards tasks with no global communication. Authors report the architecture outperforms state-of-the-art GPUs by an average of 2.54× in 10 out of the 16 benchmarks analyzed, showing superior performance whenever workloads do not require communication between near-data processing elements and do not use complex instructions such as multiplications, divisions and floating point operations in general. Chopim (Cho et al., 2020) is a near-data accelerator that focuses on the issue of managing concurrent memory accesses from the near-data device and the host. It implements near-data processing by integrating a logic die into DRAM chips and making modifications to memory controllers and data layout to allow host and accelerator to process data collaboratively. These modifications ensure data in the memory is aligned such that near-data operations over large arrays offer advantageous performance regarding execution time and energy consumption. The authors report a 2× speedup of a common Machine Learning (ML) algorithm using their approach of collaborative action between accelerator and host at separate stages of the processing. Gao et al. take a different approach by modifying the operation timings of standard DRAM memory controllers to allow for parallel computation. ELP2IM (Xin et al., 2020) also uses DRAM circuitry. It implements operations by modifying row buffer states, reducing data movement and achieving high performance regarding execution time and energy efficiency. SIMDRAM (Hajinazar et al., 2021) is a general-purpose framework for implementing complex operations in DRAM-based devices. STT-CiM (Jain et al., 2018) is an in-cell design that explores the possibility of enabling multiple word lines concurrently and executing operations directly in the memory, providing superior energy efficiency. HMC Instruction Vector Extensions (HIVE) (Alves et al., 2016) integrates vector units on the logic layer of the HMC device, enabling operations over very large vectors and taking advantage of the vault parallelism offered by the device. It adds a set of vector FUs and a register bank to the logic layer of the 3D-stacked memory. Lastly, Xie et al. (2019) use ReRAM to implement logical and arithmetic instructions that can be used to implement a novel multiplication algorithm reportedly 46% more efficient than the state-of-the-art ReRAM alternative.

3.8 SUMMARY

In this chapter, we went over a wealth of existing NDP proposals found in the literature. By identifying the behaviors for which near-data processing can be most beneficial, we were able to point out a number of application domains that benefit this type of approach. This informed our

research for related work and allowed us to categorize most of the proposals we were able to find. The work presented on this chapter has been published on the Journal of Integrated Circuits and Systems (Santos et al., 2021b).

The related work also serves to help us place our own proposal within the existing literature and identify challenges, shortcomings and opportunities we may be able to address with our architecture. More specifically, through our research of the existing literature we are able to spot a few opportunities regarding the way existing fine-grain FU-based NDP architectures function and how they can be modified and extended to achieve better performance through improved utilization of hardware resources. In the next chapter we describe VIMA, our NDP architecture proposal.

4 VECTOR-IN-MEMORY ARCHITECTURE

In Chapter 3, we discussed a number of NDP approaches and solutions, many of which report significant performance improvements regarding execution time, energy efficiency and data throughput. However, such related work fails to discuss an important aspect of modern architectures: providing precise exceptions. It can be argued that such architectures have "imprecise" exceptions, which helps them achieve significant results as they are unencumbered by the limitations providing precise exceptions would require. They are able to, for instance, issue an unlimited number of instructions to be executed near-data in parallel with no regard for possible exceptions that could arise. Should adequate handling of exceptions be considered a priority and a requirement, these architectures would likely be unable to report such large improvements.

We argue that NDP architectures can not only guarantee precise exceptions, but also that such architectures can continue to provide all overall performance improvements commonly achieved by NDP proposals while doing so. We pose the following hypothesis: **it is possible to provide precise exceptions, improved memory access and multithreading with fine-grain NDP to speedup data streaming applications**.

This thesis proposes VIMA, a general-purpose NDP architecture that takes advantage of the internal data bandwidth of 3D-stacked memories to provide maximum performance, efficiency and flexibility. VIMA can be used to improve performance of applications that stream through data in a coalescent access pattern, presenting low data reuse – or that reuse a dataset of a size that exceeds the capacity of the LLC of the system. The proposed design implements precise exceptions near the data, while also optimizing memory bandwidth usage and enabling multithreading capabilities.

Figure 4.1 illustrates the VIMA architecture. VIMA extends the host processor ISA with its own specific instructions, which are fetched and decoded as any memory instruction by the host processor, but offloaded to be executed by the near-data device at the execution stage of the processor pipeline. It adds a number of control elements to the memory controller of the system: this placement choice makes the architecture able to function with either HMC and HBM memories, which are the two most common 3D-stacked memory products available commercially. Dedicated data storage and actual processing are implemented on the logic layer of the 3D-stacked memory device.

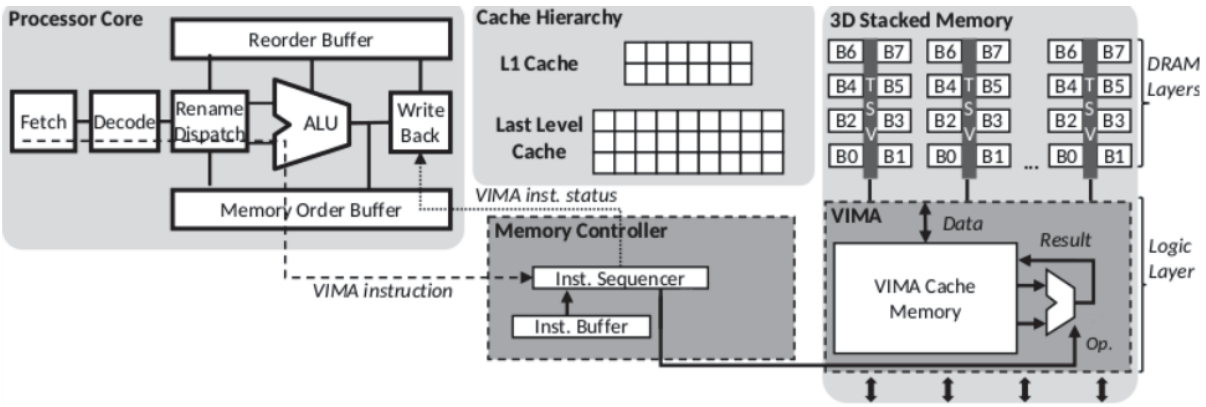


Figure 4.1: 3D-stacked memory module with our mechanism architecture.

VIMA has the following main features:

- **Near-data SIMD processing:** VIMA operates over large vectors inside the memory with each instruction.
- **ISA extension:** VIMA extends the host processor ISA, adding large vector instructions that can be used to implement any vectorizable algorithm to run efficiently within the memory.
- **Improved programmability:** the VIMA-Intrinsics library allows for easy development and deployment of applications near-data.
- **Retrocompatibilty:** the SRAM-based internal storage design and FU-based processing model of VIMA causes it to be fully compatible with further versions of VIMA without requiring existing code to be modified or recompiled.
- **Load-ahead:** our implementation of instruction pooling and guarantee of precise exception enables VIMA to safely load instruction operands out-of-order, allowing it to better exploit the internal bandwidth of the 3D-stacked device.
- **Reduced number of cores:** with a single core, VIMA matches and surpasses the execution time of a 16-thread traditional system, thus greatly reducing overall energy consumption.
- **Increased processor cache efficiency:** by performing vector operations inside the memory, VIMA avoids loading data that will not be reused into the cache hierarchy, thus reducing cache pollution.
- **Multithreading capabilities:** the flexibility features offered by the VIMA design enable near-data processing by multi-core systems.
- **Compatibility:** VIMA is compatible with both of the most common 3D-stacked memory products, HMC and HBM.

In the remainder of this chapter, we discuss our proposal in detail. Over the next several sections, we describe our contributions and address matters of programmability, data coherence, operand sizes, host system integration and processing model.

4.1 PROGRAMMABILITY AND INSTRUCTION OFFLOADING

VIMA is controlled by instructions that are added to program code and then offloaded to the near-data device at execution time. By choosing to implement this instruction offloading strategy, we avoid moving instruction fetching and decoding tasks to the memory. Instead, instruction decoding continues to be performed by the host processor, which keeps complexity of the near-data device low, and also avoids movement of instruction data between the processor and the memory, since most instructions tend to be found in the L1 cache. Depending on the size of the data operands involved in each instruction, which will be discussed on Section 4.6, such a NDP design trades moving a large amount of data from the memory to the processor for processing for only a few bytes of instruction data that will then be offloaded to be processed near the memory.

The VIMA ISA considers a subset of arithmetic and bitwise instructions commonly available in commercial SIMD extensions such as SSE, Advanced Vector Extensions (AVX) and NEON. Every instruction must stipulate the memory addresses of at least one read vector and one write vector. In case the operation the instruction refers to uses two input memory operands,

the instruction includes addresses for two read vectors and one write vector. After execution, the result of the operation is written to the write vector, which is then back to the memory when evicted from the VIMA dedicated storage.

To facilitate coding for VIMA, we provide Intrinsics-VIMA, a library in the C/C++ language that allows development using any commercial Integrated Development Environment (IDE). The library functions similarly to other existing vector instructions libraries, such as ARM NEON Intrinsics and Intel Intrinsics, and works by embedding its assembly code during compilation to optimize execution (Coorporation, 2009). Each function of the library maps directly to a specific VIMA instruction and any code written can be debugged and run on any architecture that supports C/C++ programming. This library is particularly important for this work as it enabled us to faster codify applications in high level (in contrast to assembly code) and simulate NDP that integrates FUs near data.

Code 4.1 presents the implementation of a vector sum example using Intrinsics-VIMA.

Code 4.1: 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]);
}
```

4.2 INSTRUCTION SET EXTENSIONS

In order to use VIMA, the host processor must be aware of the VIMA instruction set. Similar to proposals mentioned in Subsection 2.4.1, our architecture relies on the host to trigger instructions which then traverse the processor pipeline and are offloaded to the NDP device as needed. Table 4.1 lists all the operations supported by VIMA and

Code 4.2: Intrinsics-VIMA routine example.

```
// This routine is fully executed in any architecture
// Our simulator replaces this with a VIMA instruction
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;
}
```

The ISA was defined to implement the operations necessary to run a number of common big-data kernels, specifically those we used for evaluation of the architecture, as will be seen on Chapter 5, and also considering operations that are commonly implemented in existing SIMD extensions such as Intel AVX and ARM NEON. Code 4.2 shows the implementation of one of the Intrinsics-VIMA routines that associate with the ISA instructions. With this library we are able to code and debug VIMA in a real machine with traditional x86 system. Nevertheless, such library will also be useful for VIMA simulations as explained in Section 4.1.

Table 4.1: VIMA's proposed instruction set.

NAME	MNEMONIC	OPERATION
Addition	*add*	c[i] = a[i] + b[i];
Subtract	*sub*	c[i] = a[i] - b[i];
Absolute Value	*abs*	mask = a[i] >> shift; b[i] = ((a[i] + mask) ^ mask);
Maximum	*max*	if (a[i] > b[i]) {c[i] = a[i];} else {c[i] = b[i];}
Minimum	*min*	if (a[i] < b[i]) {c[i] = a[i];} else {c[i] = b[i];}
Copy	*cpy*	b[i] = a[i];
And	*and*	c[i] = a[i] b[i];
Or	*or*	c[i] = a[i] | b[i];
Exclusive Or	*xor*	c[i] = ~(a[i] & b[i]) & ~(~a[i] & ~b[i]);
Not	*not*	b[i] = ~a[i];
Set If Lower Than	*slt*	if (a[i] < b[i]) {c[i] = 1;} else {c[i] = 0;}
Compare If Equal	*cmq*	if (a[i] == b[i]) {c[i] = 1;} else {c[i] = 0;}
Shift Left	*sll*	c[i] = a[i] « b[i];
Shift Right	*slr*	c[i] = a[i] » b[i];
Divide	*div*	c[i] = a[i] / b[i];
Multiply	*mul*	c[i] = a[i] * b[i];
Cumulative Sum	*cum*	*b += a[i];
Move	*mov*	b[i] = a;
Load with Mask	*lmk*	if (b[i] == 1) c[i] = a[i];
Reset with Mask	*rmk*	if (b[i] == 1) {c[i] = 0;} else {c[i] = a[i];}

4.2.1 VIMA Instruction Format

Each individual VIMA instruction contains the following information: a prefix that informs the decoder the instruction is a VIMA instruction; an opcode that indicates a specific VIMA instruction; a field that identifies which core in the architecture issued the instruction; one read address field; one store address field; and one field that can store either a second read address or an immediate value. Table 4.2 shows the format of the instruction.

PREFIX	OPCODE	CORE	ADDR1	ADDR2	ADDR3/IMM.

Table 4.2: VIMA instruction format.

Each VIMA instruction is composed as follows:

- **Prefix** (1 byte): size in accordance to x86 decoder style, informs the decoder the instruction is a VIMA instruction.
- **Opcode** (1 byte): instruction identifier; used to determine one instruction implemented in the VIMA ISA.
- **Core** (1 byte): core identifier; used by VIMA to indicate which core should be notified of the instruction status upon completion or exception.
- **Address 1** (8 bytes): memory address from which the first read vector operand will be loaded.
- **Address 2** (8 bytes): memory address to which the write vector operand will be stored.

- **Address 3/Immediate Value** (8 bytes): memory address from which the second read vector operand will be loaded, or immediate value.

4.3 ADDRESS TRANSLATION

As discussed previously, VIMA instructions cause at least 1 load operation and 1 store, each being the size of the operand width being implemented. The memory addresses must be translated by the MMU, which means in our system the TLB, and be applied every check as any regular memory operation. Due to the size of the vectors, this requires the TLB to support very large pages, which it does for most modern systems (Kwon et al., 2016).

4.4 DATA COHERENCE

As a VIMA instruction makes it way through the processor pipeline and reaches the execution stage (as pictured on Figure 4.1), it is ready to be offloaded for near-data execution. However, when considering a near-data processing architecture, one must manage how near-data instructions and tasks potentially interact with the rest of the system, since the NDP accelerator and the host processor share the same memory space. This means addressing, when designing a near-data processing device that shall act as a co-processor, how the system will react regarding its memory subsystem when tasks and/or instructions are delegated to the co-processor, so data coherence is maintained.

One way to manage this issue is to guarantee that there is no intersection between the data stored in the host processor cache hierarchy and the data being operated on by the near-data processing device at any time. This can be achieved by observing the memory address ranges that shall be accessed by a near-data instruction or task and assuring that no system element risks accessing obsolete data. In practice, this means going through the entire cache hierarchy, invalidating all lines containing data the near-data instruction will touch and writing back to memory all data modified by the host processor. Thus, it is enough for the programmer to guarantee that the host will not attempt to access data in the same address range the near-data processing element is operating on before its results are written back to the memory, data consistency can be guaranteed and race conditions avoided.

This large amount of cache line invalidation, however, severely hinders the ability of architecture to provide a performance improvement, as is the case with any NDP effort. Guaranteeing cache coherence is largely regarded as one of the main issues near-data architecture designs face (Ahn et al., 2015b; Boroumand et al., 2019, 2016).

One feasible, less onerous alternative to guaranteeing cache coherence is to use flush instructions, which are present in most host processor ISA and can be used by applications (Drebes et al., 2020; Santos et al., 2021a). Thus, were a host to modify a memory address that gets subsequently accessed by a VIMA instruction, a flush instruction could be emitted to make sure the modified data gets written back to the main memory to ensure a near-data operation will not attempt to operate over obsolete data.

Of course, should this operation apply over the entire cache, it could remove data outside the scope of operation of the VIMA instruction, causing cold cache effects. However, by utilizing data access patterns indicated by the compiler/programmer, a selective flushing behavior can be achieved. Flush instructions can be added to be automatically generated by the compiler, avoiding burdening the programmer with this task: should a VIMA instruction access a specific data element that has been recently accessed by the host and is stored in the cache, the compiler ensures coherence by including flushing instructions in the program code to flush all related

cache lines. Nevertheless, this approach is considered non-coherent, as the hardware does not provide transparent coherence (i.e. the coherence is maintained by the software). Moreover, there can be no guarantee that the programmer will be careful enough to avoid all data collisions or race conditions the compiler is not able to detect.

Another possible strategy is to determine, within some specific range address of the systems' memory space, that any data placed in it shall be ignored by the cache memory subsystem. Thus, whenever any reads from or writes to an address within that specific range, the data is not moved to the cache memory, forcing a direct load or store to the main memory. Should the near-data portions of an application restrict themselves to dealing only with data within that space, cache coherence issues would be avoided by default. Such approach could be easily constructed using techniques similar to non-cacheable instructions already present on modern processors.

To illustrate the impacts of different approaches to cache coherence, we ran an experiment. It simulated running the same algorithm over the same input dataset on VIMA, considering different cache coherence strategies. The algorithm is a simple *memory copy application*, which copies the contents of one position in the memory to another. The strategies we simulated are as follows:

- Coherent with forced flush: in the first strategy, the cache hierarchy is checked for all operands in a VIMA instruction before it can be processed near-data, writing back all modified lines and invalidating all hits.

- Non-Coherent NDP: the second strategy assumes a compiler-aided situation in which the compiler, leveraging data access pattern information, generates flush instructions for around 11.2% of all the data being referred to by VIMA instructions. This collision rate was chosen according to related work (Boroumand et al., 2019) that has looked into average data collision rates between CPU threads and NDP accelerators. All flushing happens before the instruction is offloaded for near-data processing.

- Coherent with non-cacheable region: lastly, the third strategy considers all data handled by VIMA is kept within a non-cacheable region of the memory (Ahn et al., 2015a), thus causing VIMA instructions to completely bypass the cache hierarchy.

We assume 8192 B vector operands for all VIMA instructions and used input sizes ranging from 8 MB to 64 MB. Figure 4.2 plots the results of this experiment. Results are normalized to a 16-thread x86 baseline, meaning results over 1 indicate an advantage over the baseline. Details regarding the simulated systems are detailed in Chapter 5.

With the *coherent with forced flush* strategy, considering a 64 B cache line in the cache hierarchy, a single VIMA 8192 B vector requires checking for 128 cache lines. In a worst case scenario, the system would be forced to withstand the latency of accessing and writing back all 128 lines to the memory per vector operand (i.e. 8,192B / 64B) before issuing each instruction. The latency caused by this comprehensive checking and writing back of cache lines becomes overwhelming and negates any performance that can be gained from near-data execution. We can expect to avoid a large portion of this latency by using the *non-coherent* strategy, but would be relying on the programmer to write code that avoids most collisions and on compiler analysis to be able to detect when collisions happen. In the worst case scenario, the system would be forced to withstand the latency of accessing and writing back the entirety of the vectors to the main memory. Meanwhile, considering an architecture that completely bypasses the cache hierarchy, improvement in performance for the workload used in the experiment, a memory copy operation, yields a 6× reduction in execution time when looking at the largest input size considered. When considering the strategy based on flush operations, the reduction in execution

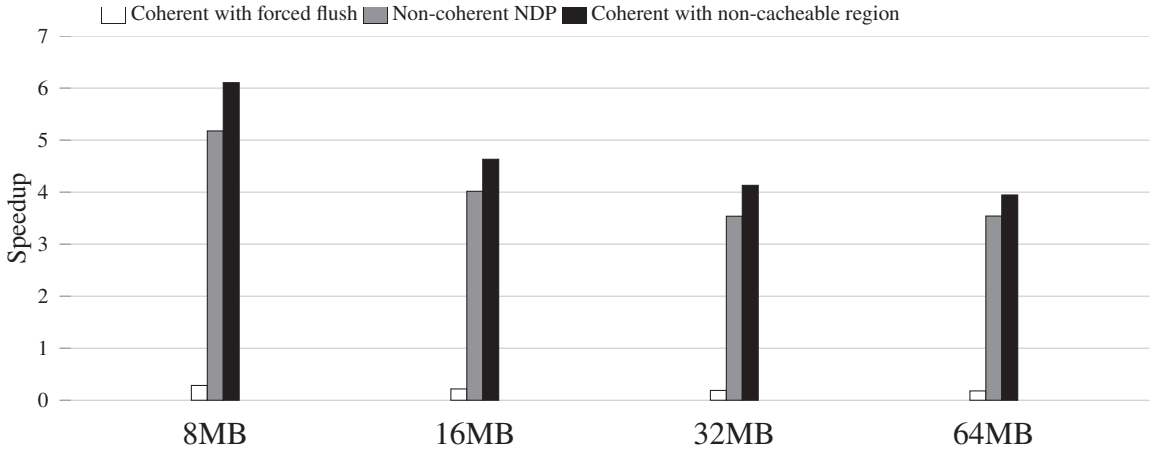


Figure 4.2: Execution time speedup results executing *memory copy application* comparing VIMA with several approaches to cache coherence. Results are normalized to a 16-thread x86 baseline.

time is slightly smaller, at just over 5×. Naturally, our choice to consider that only 11% of all data is accessed by both host and VIMA during the simulation is based on an average (Boroumand et al., 2019) and the performance of any NDP architecture will suffer when using this strategy if this collision rate is high. We thus choose to consider that the memory area that VIMA accesses is deemed non-cacheable (Ahn et al., 2015a) for all subsequent experiments. It should be noted, however, that VIMA is orthogonal and fully compatible with recent near-data architecture cache coherence proposals, such as CoNDA (Boroumand et al., 2019), which reports positive results. Implementing their cache coherence scheme for VIMA is one avenue for future work.

4.5 PLACEMENT IN THE SYSTEM

DRAM is the memory technology most commonly used for main memory in traditional computer systems. As applications start to require access to larger and larger amounts of data and the number of processing cores in traditional systems increase, DRAM technology had to follow suit to handle the increased pressure to which the main memory of most systems was subjected. This response came in the form of an increase in the number of channels, memory controllers and parallelism in the data access to ensure memories were able to deliver data at adequate rates. However, as modern memories evolve and provide ever higher bandwidths, new issues emerge regarding the ability of systems to take advantage of such bandwidths these memory chips are now equipped to provide (i.e. the logic division into memory vaults). Although it can be argued that wider data buses and larger cache hierarchies could be used to do so, these are not sustainable solutions as their area and power requirements also increase with size, especially with the end of Moore's Law and Dennard scaling.

DRAM-based NDP systems then take an approach that attempts to bypass most of the limitations that hinder memory bandwidth. To avoid barriers such as narrow buses or off-chip communication latencies, these systems try to access data as close as possible to where it is stored in the system. As mentioned in Chapter 3, several existing proposals do so by modifying the circuitry and/or behavior of the very memory cells that store the data to perform computation, thus working as physically close as possible to the data. This is, although energy efficient and fast, very limiting in terms of the types of operations that can be performed over such data. Therefore, several NDP proposals turn then to the next best thing: accessing data in the DRAM row buffers.

In the case of the 3D-stacked memory technology devices, having close access to the DRAM row buffers yields a benefit that is even more pronounced due to the significant increase in bandwidth offered by the hardware under these conditions, allowing for superior data throughput rates in comparison to traditional memories. The HMC 3D-stacked memories are split into up to 32 independent vaults, each of which has its own row buffer, supplying up to 256 B per access (Hybrid Memory Cube Consortium, 2014), meaning up to 8192 B can potentially be accessed in parallel per cycle, considering a sustained pipelined scheme of accesses. HBM memories, on the other hand, have up to 16 independent vaults, wider row buffers and support 128 B requests, meaning different memory architectures will enable different possibilities.

To take advantage of such possibilities, VIMA is placed on the logic layer of a 3D-stacked memory device, a placement that allows it to both read and write quickly from and to DRAM banks. Figure 4.3 illustrates the position of the VIMA device relative to the memory device. In the case of the HMC, it could be placed coupled to the cross-bar switch. For the HBM, it would be placed in the interposer. In both situations it would have access to all the vaults, in order to access any operand independently of its allocation.

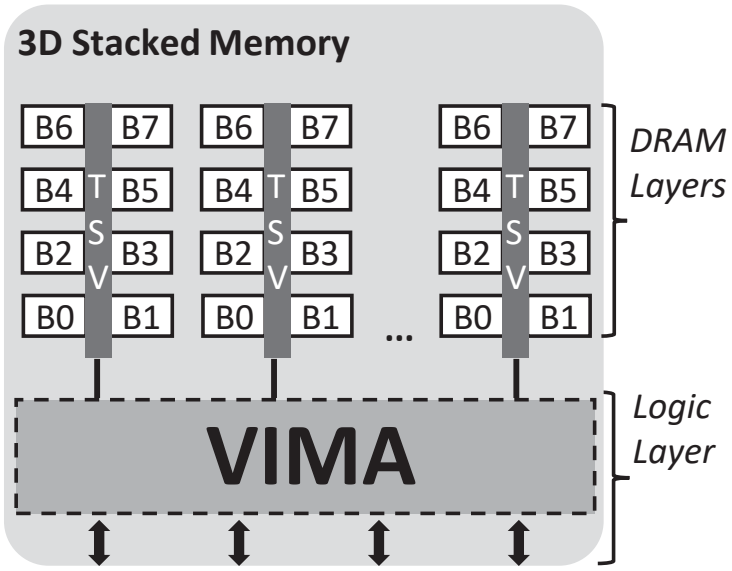


Figure 4.3: Position of the VIMA device relative to the 3D-stacked memory.

4.6 VECTOR SIZE

Multithreaded systems traditionally benefit greatly from their ability to fetch and process data in parallel. Since each core is equipped with its own set of functional units and register banks, such systems are able to issue a large number of memory requests in parallel, applying increased pressure to the main memory and using much of its bandwidth. On the other hand, the functional units-based near-data architecture like VIMA, in order to favor simplicity and energy efficiency, is unable to behave like a superscalar processor. Therefore, to provide an execution time performance improvement over such systems, the vector size used by the device must be large enough to match or surpass such levels of parallelism by leveraging as much of the memory bandwidth as possible. Nevertheless, the vector size also impacts on the amount of VIMA instructions the processor needs to trigger to our architecture, which also impacts energy and time. The smaller the operand size, the more instructions the processor must trigger to fully process a given dataset.

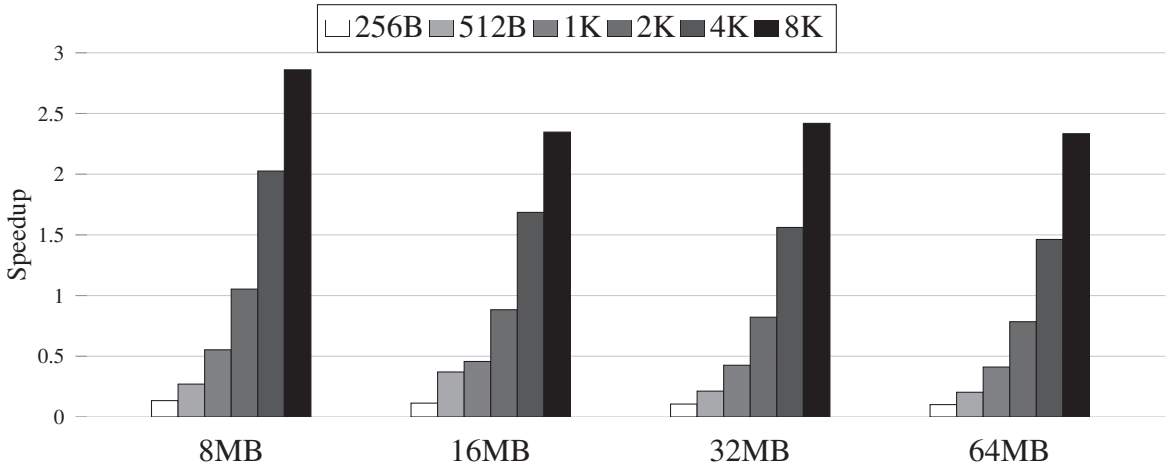


Figure 4.4: Execution time results executing *vector sum application* comparing VIMA with various vector widths normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

To leverage the parallelism of the 3D-stacked memory chips, VIMA instructions must operate over adequately sized operands. Figure 4.4 shows the results of a simple experiment that illustrates this point. For this, we assume the underlying 3D-memory stacked device used by VIMA is the HMC 2.1, which has 32 independent memory vaults and a row buffer size of 256 B. Considering a simple vector sum application, we ran simulations with several different vector sizes, ranging from 256 B (accessing one memory vault per vector operand) to 8192 B (accessing all 32 memory vaults in parallel per vector operand), operating over datasets ranging from 4 to 64 MB in size. The results on the figure refer to relative performance compared to a 16-thread x86 baseline, meaning values over 1 imply an improvement in performance and values under 1 imply performance degradation. As the difference in performance is reduced as the dataset grows, likely due to less multithreading overhead in the baseline, it is clear that the near-data architecture only provides an improvement over the 16-core x86 baseline at large dataset sizes if the width of its vector operands is sufficiently large. In our experiments we consider that VIMA implements only one specific vector width, but exploring the possibility of supporting configurable vector widths to provide a choice of width and optimize benefits according to the application is one avenue for future work.

4.7 DATA STORAGE

After a VIMA instruction arrives at the memory controller, its operands must be fetched from the memory and placed in VIMA's dedicated cache memory before the operation is applied to them. This cache is fully associative following a Least Recently Used (LRU) replacement policy: if there are no empty spaces available in the cache, the least recently used vector currently in it gets evicted to clear up space for a new one. In case it has been modified, it is written back to the memory. Although we consider a fixed capacity for the dedicated cache for our experiments, the memory space can be expanded in further versions of this architecture as needed. In fact, this expansibility is one of the reasons why we chose to use a cache memory as opposed to a scratchpad memory or register bank.

When choosing which type of memory to use in such a design, one must consider the benefits and drawbacks of their choice. For instance, if we chose to use a register bank for storage in our VIMA architecture like is done in a number of similar designs (Alves et al., 2016; Santos et al., 2017; Tomé et al., 2018), this would have several consequences. Especially in comparison

to a cache memory, using a register bank has lower area and energy consumption requirements, while providing faster access to the stored data. On the other hand, it also invites a number of limitations. For instance, if memory capacity is increased in the architecture, using a register bank would require all VIMA code to be recompiled in order to access the new space, as the compiler would have originally considered only the existing registers at compilation time. With a cache memory, any improvement in capacity would translate into improved performance for data reuse applications transparently. Another issue that pops up with register banks is how to manage processing when multithreaded applications are considered. Using a register bank as memory would require a locking/unlocking mechanism to avoid race conditions between threads trying to use the same register within the near-data device, which a cache memory avoids.

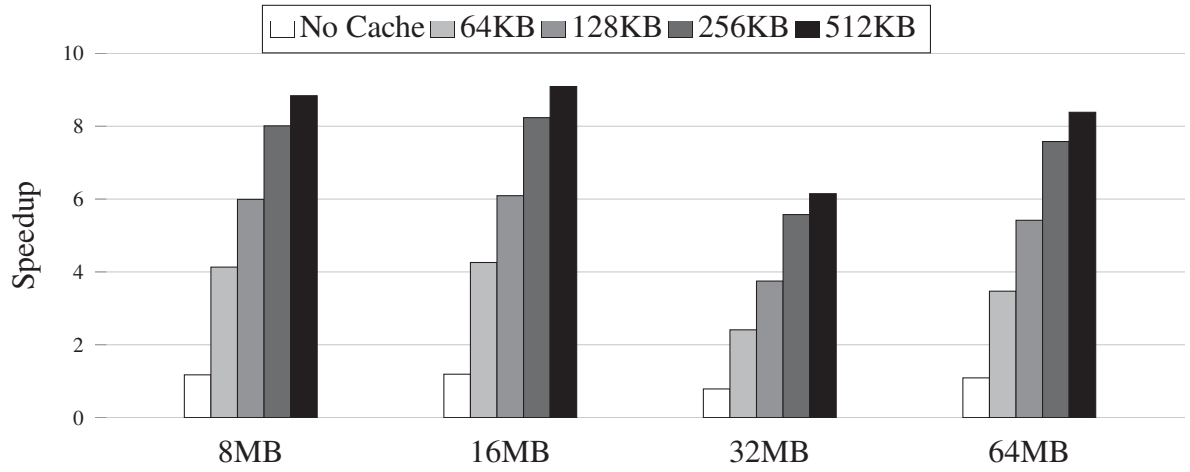


Figure 4.5: Execution time results with VIMA running *bloom filter application* with varying data storage, 8192 B vector operands and a HMC 2.1 memory chip. Values higher than 1 indicate improvement in performance over the baseline.

Naturally, using a cache memory also makes data reuse much simpler. Figure 4.5 compares VIMA with its data reuse capabilities and a version that is unable to reuse data and loads data from the memory for every instruction. The application used for this experiment is a bloom filter computation. The bloom filter is a data structure commonly used for pattern matching and it is accessed very often as the nature of it functions, thus causing lots of opportunities for data reuse. As can be seen in Figure 4.5, performance improves significantly when a cache memory as used.

Performance also improved significantly as the size of the cache storage increases. When comparing results of the experiment with no cache and the one with a 64 KB cache, execution time is reduced by 80%, which shows that the application uses the cache very efficiently and therefore greatly benefits from it. Between the 64 KB and 256 KB sizes, execution time is further reduced by about 30% with each increase. Starting at 512 KB, however, the improvement on execution time is less than 9%, suggesting there will be diminishing returns from increasing storage space any further. For the experiments found on Chapter 5 we consider VIMA with a 256 KB cache, meaning that when considering a VIMA operand vector of 8,192B, this cache provides 32 different cache lines.

4.8 PRECISE EXCEPTIONS

Upon adding NDP capabilities to a system, we need to consider that, even if its host processor has a single processing core, the system becomes multiprocessed, with two separate processing

elements (the traditional core and our proposal) sharing the same memory space. As VIMA must also change the architectural state of the system, we must carefully design its function to avoid consistency issues.

Even with multiple processors in a system, programmers will assume a sequentially consistent memory model. This assumption comes with three main expectations: (i) that memory operations are executed one at a time in atomic fashion; (ii) that each processor respects program order when issuing memory requests, which then also affect the memory state in the same order; and (iii) that, for every memory location, any load operation will return the value from the last store operation done to that location. Although architectures do not necessarily adhere strictly to such orderings during actual processing, which allows them to optimize performance, execution results must appear to the programmer as if they did. This consistency requirement is expected of any system with multiple separate processing elements, regardless of the position of such elements in the system. Consequently, providing precise exceptions is important to enable adoption of an NDP architecture as it guarantees the sequential consistency memory model users expect.

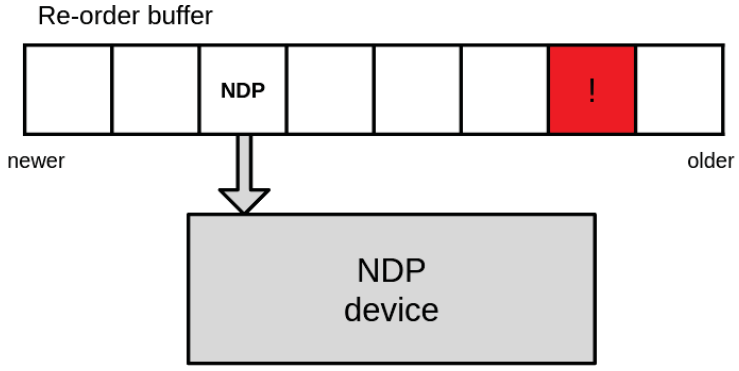


Figure 4.6: NDP instruction is offloaded to device and x86 instruction raises exception.

Current state-of-the-art NDP architecture proposals such as HIVE (Alves et al., 2016) do not guarantee precise exceptions, leaving the task of maintaining sequential consistency up to the programmer writing the application that accesses the NDP resources. By doing so, these systems would risk producing an inconsistent state in the system in situations such as the one illustrated in Figure 4.6. Should an NDP instruction be offloaded for near-data execution and then an exception be raised by an older traditional instruction in the host processor's re-order buffer, such an architecture would have no way to guarantee that no modifications were made to the memory before this exception was handled. Although the NDP-specific instruction would be flushed out of the re-order buffer as the system reacted to the raising of an exception, it still may have modified the architectural state of the system, causing an inconsistent memory state, for instance.

A simple way to guarantee precise exceptions and maintain system consistency is to only allow the NDP device to handle one instruction or task at a time, when that instruction becomes the oldest in the host processor's re-order buffer (i.e. head of the ROB), as can be seen in Figure 4.7. By making sure only one instruction or task is executing in a separate processing element, the host processor can wait for its execution and status report before committing its results to the architectural state. Should an exception arise during the execution of the NDP instruction, the processor treats it like any other exception, with a new exception code and entry in the Interrupt Descriptor Table (IDT) (Ahmed et al., 2013) for NDP exceptions. While this approach guarantees precise exceptions, it also requires the NDP to be idle for some time. Between the moment the NDP reports the instruction status and the moment another NDP

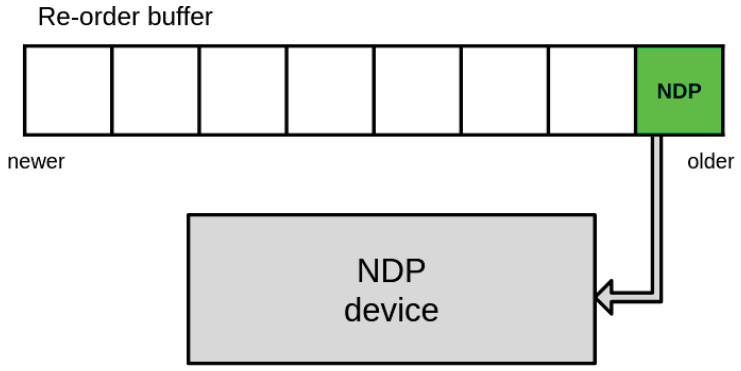


Figure 4.7: NDP instruction is offloaded to device once it becomes the head of the re-order buffer.

instruction moves forward in the host processor pipeline and reaches the memory device, the NDP will stay idle as it awaits further tasks. Were our proposal to follow this approach, VIMA would likely fail to achieve performance improvements that are comparable to state-of-the-art related work (Alves et al., 2016; Tomé et al., 2018). Since they do not guarantee precise exceptions, they are able to issue multiple instructions for near-data execution at once.

To avoid this issue, and thus better exploit the capabilities of the underlying main memory while still being mindful of precise exceptions, we propose devising a system that allows buffering of multiple instructions on the device. Were the device able to pool instructions, it could avoid most idle time, as it could move on to the execution of the next instruction as soon as the previous one finishes. Figure 4.8 illustrates the idea.

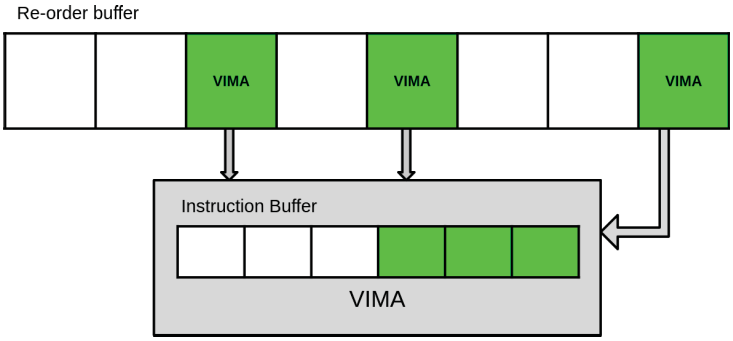


Figure 4.8: VIMA-specific instruction buffer stores multiple instructions at once.

However, should an exception arise during near-data execution of an instruction, it would have to be able to take steps to react to this (i.e. squashing all the instructions in both the processor's re-order buffer and its own buffer that are newer than the instruction that raised the exception). Thus, the responsibility of maintaining precise exceptions whenever near-data processing takes place is shared between the host processor to the NDP device. Figure 4.9 illustrates this situation.

Enabling instruction pooling requires making a few modifications to the architecture. First, to guarantee overall data coherence in the system, it must ensure that no instructions between NDP instructions in the program change the memory hierarchy state. Were this not observed, the NDP device would risk fetching outdated data from memory and subsequently writing erroneous results to the memory.

Second, the NDP device would have to maintain its own instruction buffer of its specific instructions, which would require adding circuitry to the device. The device would also be responsible for flushing its own instruction buffer and any associated stored data in case of an

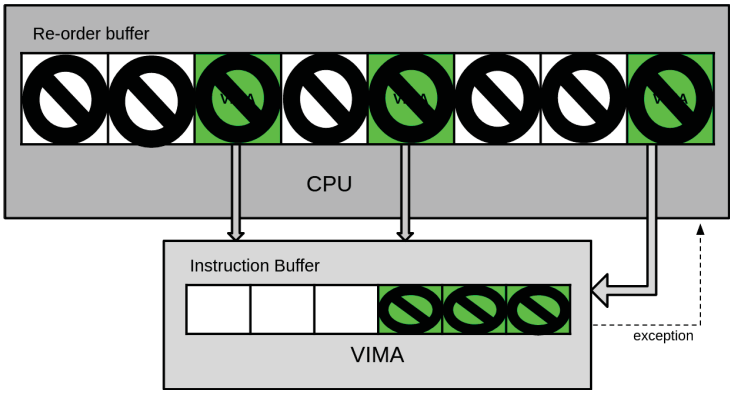


Figure 4.9: Exception is raised and VIMA instructions in the specific buffer are flushed along with all instructions in the host processor's re-order buffer.

exception. To achieve instructions pooling with VIMA while maintaining precise exceptions, we add two elements to the memory controller of the system: (i) an instruction sequencer, which controls instruction order and (ii) an instruction buffer, which stores VIMA-specific instructions. VIMA instructions are stored in the instruction buffer and handled by the instruction sequencer, who controls which instructions are ready to be executed next.

Should an instruction arrive when the instruction buffer is empty and, therefore, the device is at an idle state, it starts being processed right away by issuing requests to the memory for its operands. The instruction sequencer observes the memory addresses for the operands required by the instruction and communicates with the dedicated cache memory to check for whether the required data is present. It is thus able to inform the memory controller what data addresses to request from the main memory. As memory requests made by VIMA are fulfilled, the data is stored in the dedicated cache in vectorized fashion. These requests are labeled as VIMA requests and its data are stored in the near-data dedicated cache instead of being returned to the host processor.

Once all operands of an instruction are successfully loaded and placed in the cache memory, an instruction is deemed ready for execution, which happens immediately thereafter if the instruction is the oldest in the instruction buffer. Results are written to the store operand specified in the instruction, also in the cache memory.

4.9 THE LOAD-AHEAD MECHANISM

The ability to pool instructions and also guarantee precise exceptions allows VIMA to further optimize its usage of its underlying memory architecture. While it is no longer forced to be idle between instructions, its knowledge of buffered instructions can be leveraged to exploit data throughput available in the memory even more efficiently. Considering its cache-based storage, VIMA can safely prefetch operands of instructions in its buffer even if the instruction is not currently the oldest in store. Although the actual execution and committing of VIMA instruction results to the memory happens strictly in order, whenever the operands of multiple instructions do not overlap, data fetching for operands of distinct instructions can happen out-of-order. We call this the **load-ahead mechanism**.

Figure 4.10 compares a 16-thread x86 baseline to VIMA both with and without our load-ahead mechanism. For this experiment we use 8192 B vector operands and a HMC 2.1 chip as underlying 3D-stacked memory. Results refer to the speedup over the baseline achieved at four distinct dataset sizes when processing a simple application that sets every data point in a large

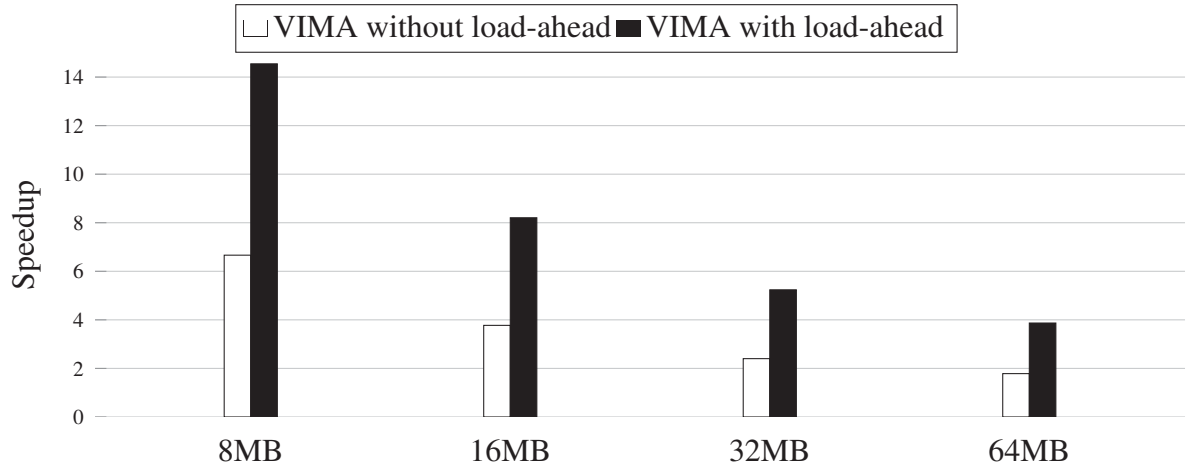


Figure 4.10: Speedup of VIMA over baseline running *memory set application* with 8192 B vector operands and a HMC 2.1 memory chip. Values higher than 1 indicate improvement in performance over the baseline.

vector stored in the memory to the same integer value. While VIMA without the load-ahead mechanism outperforms the 16-thread baseline even at the 64 MB dataset size, the difference with the load-ahead mechanism is sizeable, reaching a 4× execution time improvement over the multithreaded baseline even with a 64 MB input dataset size. For this experiment, VIMA manages to extract an average 267 GB/s data throughput from the memory when considering the load-ahead mechanism, and only an 129 GB/s average without it.

4.10 INSTRUCTION EXECUTION

Actual execution of VIMA instructions happens as soon as operands are fully available in the dedicated cache and the instruction is the oldest in the VIMA instruction buffer. Thus, although operand fetching can be done out-of-order, instruction execution and data committing is kept strictly in-order.

VIMA uses a set of 64 512-bit vector units to process data, which makes it able to process up to 2,048 bytes at a time. In case the vector operands used are larger than this size, they are divided into as many chunks as necessary for execution in a pipelined fashion. Using such a reduced number of functional units, although it forces us to split the data in several parts for processing, has straight-forward benefits, like a smaller area requirement and low energy consumption. A smaller number of functional units also causes the project to be more physically viable. Since data access for processing will be happening from the dedicated cache memory, the functional units must connect to it. Were we to use an amount of functional units that is able to process, for instance, an entire 8192 B vector at once, this would require 65,536 connections between FUs and cache memory, which seems to be impractical. With a smaller number of FUs, although we make more accesses to the SRAM chip per vector, we require much fewer connections. Nonetheless, we consider that further low level integration analysis is out of scope of this work.

Interestingly, a reduced number of functional units does not impact performance very much. This happens because of the aforementioned load-ahead mechanism used for fetching of data, which causes the execution time of VIMA instructions to be masked by the loading of other instructions' operands, effectively hiding the added latency of processing each large vector in chunks. In our simulated experiments, the difference in performance between this and a version

of VIMA with enough functional units to process an entire 8192 B vector at once, improvement in performance regarding execution time was less than 3%.

4.11 MULTITHREADING

The main issue in enabling multithreading for near-data architectures is the matter of making sure distinct threads do not modify data fetched by other threads when instructions or tasks offloaded by separate cores co-exist in the near-data device, which is very limited and thus very prone to collision. Similar architectures to VIMA, such as HIVE (Alves et al., 2016) and HIPE (Tomé et al., 2018), rely on a register bank for storage, and thus have to control this by implementing a locking and unlocking mechanism to guarantee only one thread is granted access to the near-data capabilities at a time, meaning that multiple threads are serialized. By using a cache memory with a replacement policy that is automatically handled by the device we can guarantee that multithread execution can happen safely with VIMA. Thus, although at the expense of slightly longer data access latency, the issue is resolved and a parallel multithread approach to near-data processing becomes viable. Naturally, this assumes some care from the programmer/application side, which should be coded to partition its data as safely as possible to avoid multiple threads accessing the same data portions at the same time. As long as the memory regions accessed by each core are carefully managed by the application so as to not overlap each other, multithread execution can take place safely.

As previously discussed, when dealing with data streaming applications, the main limitation to performance systems face is data throughput and we mentioned in Section 4.6 that SIMD-based NDP systems achieve high data throughput based on the size of the vector operands used in its instructions: the larger the operand, the more requests are made to the memory to fetch them. By utilizing a large enough vector operand width, the architecture is able to sustain pressure to the memory even with a reducer number of instructions. However, the requirement of having to split input data into such large vectors to use VIMA can be limiting, impractical or even unfeasible for some applications. With multithreading, VIMA is able to extract data throughput and good execution time performance even at reduced vector sizes. The combination of the **load-ahead mechanism** with the use of a dedicated cache memory causes VIMA to be uniquely able to apply adequate pressure to the memory even with a smaller vector operand width.

4.12 REACTING TO X86 EXCEPTIONS

When considering all the novel possibilities VIMA offers, we must be careful when dealing with any exceptions that should arise in order to avoid creating an inconsistent system state. At any point when VIMA resources are being used, we must take steps to maintain sequential consistency should an x86 exception arise. When this happens, the processor core at which the exception was raised must signal VIMA that the issue has happened. While the processor starts its default exception handling behavior, VIMA must react by flushing out all instructions in its instruction buffer that were emitted by the core at which the exception happened. In case any operand data of the flushed instructions had already been fetched from the memory and placed in the dedicated near-data cache, all such cache lines must be invalidated.

Since the cache memory can transparently accessed by multiple threads, we must acknowledge the possibility of data reuse between threads. When accounting for exceptions, one possible issue is of there being reuse of data that was fetched and modified by an instruction that could still be flushed because of an exception, which would cause an inconsistency. To avoid this issue, we establish that data in the dedicated VIMA cache must only become available for reuse

by an instruction once the instruction that originally caused the data to be fetched from the main memory has been successfully committed and removed from both the host processor re-order buffer and the VIMA instruction buffer.

When a VIMA instruction causes an exception, VIMA must signal the host processor by reporting the exception through its status. As soon as the VIMA instruction becomes head of the re-order buffer, the default exception handling behavior of the system must be used to recover from the exception, including all communication that should happen with NDP device.

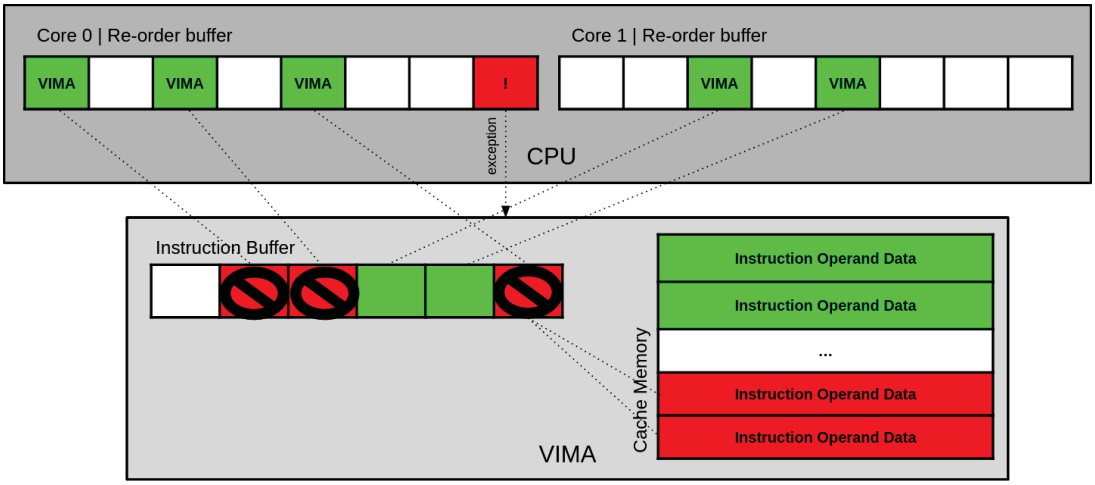


Figure 4.11: Exception is raised and VIMA instructions from the core that raised the exception. All related operand data in the cache memory is invalidated.

Figure 4.11 shows an example of this situation. While two processing cores have both issued instructions for near-data execution, an exception is raised by an x86 instruction at Core 0. The processing core signals VIMA that an exception has been raised, which means all instructions in its re-order buffer will be flushed, including any VIMA instructions that are present in it. Since each VIMA instruction holds the information of which core issued it, VIMA is able to react by removing from its instruction buffer all instructions that were issued by Core 0 and also invalidating all data that have been fetched into the dedicated cache memory to service any such instructions.

4.13 SUMMARY

In this chapter, we presented VIMA, our NDP architecture proposal. We discussed how VIMA deals with the several integration challenges faced by NDP solutions and established its placement in the system. Our design choices regarding operand sizes and dedicated storage were described and justified with simulation experiments. We also described the main contributions our design offers compared to prior work: the implementation of precise exceptions, the load-ahead mechanism and the multithreading capabilities. In the next chapter we evaluate our solution by simulating execution of several big-data kernels on VIMA under a number of possible configurations.

Portions of the work presented on this chapter has been published at Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) (Cordeiro et al., 2021; Santos et al., 2022b), and International Symposium on Performance Analysis of Systems and Software (ISPASS) (Santos et al., 2022a). A research paper featuring portions of this work is currently under review at The Journal of Supercomputing.

5 EVALUATION

In this chapter, we describe our methodology and share results for our evaluation of VIMA. The baseline architecture considered in our experiments mirrors the Intel Skylake microarchitecture and includes Intel AVX-512 extensions. We henceforth refer to it as **x86**. Since traditional systems implement several mechanism that help them better exploit memory resources, such as multiple superscalar cores, SIMD instructions and out-of-order execution, they are able to apply a significant amount of pressure to the memory, thus taking advantage of a large portion of the data throughput these devices are able to provide. For this reason, we consider a 16-thread system as our baseline, constructing a tough case against our proposal.

We consider for our experiments that VIMA runs on a single-core system (except on section 5.7 when we evaluate a multithreading system using VIMA) including all necessary architectural modifications and otherwise mirrors an individual core of the baseline. We performed all simulation experiments using Ordinary Computer Simulator (OrCS), an open-source cycle-accurate simulator based on SiNUCA (Alves et al., 2015c). Table 5.1 shows all simulation parameters for the baseline and VIMA's host processor. Table 5.2 shows all VIMA-specific parameters.

Table 5.1: Baseline system configuration.

OoO Execution Cores
16 core @ 2.0 GHz, 32 nm; Power: 6W/core; 6-wide issue;
Buffers: 40-entry fetch, 128-entry decode; 168-entry ROB;
MOB entries: 72-read, 56-write; 2-load, 1-store units (1-1 cycle);
4-alu, 1-mul. and 1-div. int. units (1-3-32 cycle);
2-alu, 2-mul. and 1-div. fp. units (3-5-10 cycle);
Branch predictor: Two-level GAs. 4096 entry BTB; 1 branch per fetch;
L1 Inst. Cache
64 KB, 8-way, 4-cycle; 64 B line; LRU policy;
Dynamic energy: 194pJ per line access; Static power: 30mW;
L1 Data Cache
64 KB, 8-way, 6-cycle; 64 B line; LRU policy;
Dynamic energy: 194pJ per line access; Static power: 30mW;
L2 Cache
1 MB, 16-way, 34-cycle; 64 B line; LRU policy;
Dynamic energy: 340pJ per line access; Static power: 130mW;
LLC Cache
16 MB, 16-way, 52-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 cy.;
8 B burst width at 2.5:1 core-to-bus freq. ratio; Open-row policy;
DRAM: CAS, RP, RCD, RAS, CWD lat. (9-9-9-24-7 cycles);
Avg. energy per access: x86:10.8pJ/bit; Static power 4W;

Table 5.2: VIMA system configuration.

VIMA Processing Logic
Operation frequency: 1 GHz; Power: 3.2W;
32 int. units: alu, mul. and div. (8-12-28 cy. per chunk pipelined);
32 fp. units: alu, mul. and div. (13-13-28 cy. per chunk pipelined);
VIMA cache: 256 KB, fully assoc., 4-cycle per access;
Dynamic energy: 194pJ per line access; Static power: 134mW;

5.1 ORDINARY COMPUTING SIMULATOR

When designing and evaluating new computer architectures, systems are often too complex for a strictly analytical model. At the same time, it is impractical and expensive to create physical prototypes for real world validation (Jain, 1990). Thus, in order to evaluate proposals of new systems, computer architecture researchers turn to simulation environments. Trace-driven simulators represent a portion of such environments: they focus on the behavior of the microarchitecture under traced applications while not requiring the actual execution of instructions to be simulated. They receive as input a description of a single execution of a program, usually containing the instruction flow that was observed during a real execution of said program. Such trace-driven simulation is based only on the application behavior, presenting determinism during execution, avoiding thus multiple executions to mitigate influence of the operating system on the measures. Such input can be generated automatically by binary instrumentation tools that are executed alongside a program for which one wishes to generate trace files for simulation, or manually generated by researchers according to specific needs.

For the simulations needed for this work we used OrCS, an open-source trace-driven simulator that implements the x86_32 and x86_64 ISA and that we extended to implement the VIMA ISA. OrCS is based on Simulator of Non-Uniform Cache Architectures (SiNUCA), another existing open-source and validated simulator (Alves et al., 2015c; Alves, 2014). All input traces are generated with the OrCS trace generator, which is based on Pin, an instrumentation tool from Intel (Bach et al., 2010). The trace generator tool uses Pin routines to observe the execution of applications and generate trace files for simulation with OrCS. When running code that includes Intrinsics-VIMA routines, the trace generator replaces the description for the x86 assembly code with corresponding VIMA code for simulation purposes only. Figure 5.1 (Cordeiro, 2020; Cordeiro et al., 2017, 2021) illustrates one such replacement. With this solution, although our experiments require simulating an ISA that is not implemented by an existing system, we avoid the need to write simulation traces by hand. Thus, we avoid bugs while increasing variety of the migrated applications to NDP.

For our experiments we implemented all workloads for both the 16-thread x86 baseline and VIMA systems in C++ and generated simulation traces with our Pin-based trace generator. These traces were then used as input for OrCS, which outputs a comprehensive list of results regarding the simulation statistics for the system being simulated. For each experiment, we simulated both the baseline and VIMA versions of each workload and input size, observing the differences in results. All code we used for our experiments is available on our repositories[1][2][3].

Here, we focused mainly on the number of cycles each experiment took to finish, from which we derived the execution time of each experiment. We also paid close attention to the

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

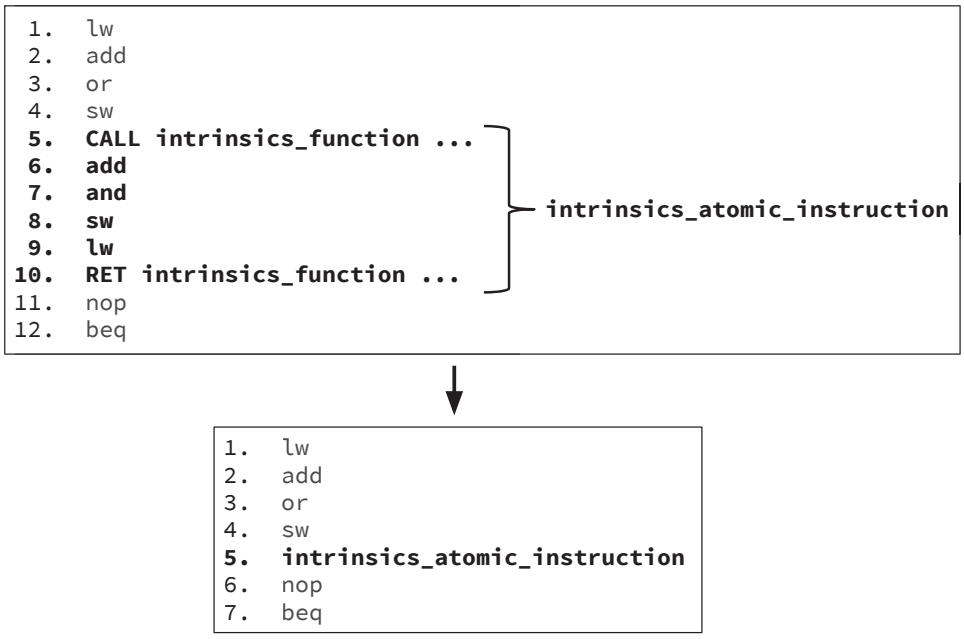


Figure 5.1: Example of x86 assembly replacement (Cordeiro, 2020).

results regarding the memory subsystem, including cache hierarchy hits and misses, and accesses to DRAM. All such stats were then used to generate our energy consumption estimates. We share our experiments results later in this chapter.

5.2 MEMORY DEVICES

Theoretically, VIMA is able to function with any version of 3D-stacked memory described on Section 2.5, observing their features and limitations. We must note, though, that the organization of the devices directly impacts VIMA performance. Since VIMA is a monolithic device that moves data out of the vaults of the 3D-memory, we expect performance to be superior on memory devices that favor vault parallelism, as opposed to bank parallelism. Here we consider that the memory controller maps the least significant address bits to vaults and most significant bits to memory banks (similar to what occurs on multi-channel systems with DDR-x devices). NDP strategies such as RVU (Santos et al., 2017), which place independent processing elements on each individual vault of a 3D-stacked memory should be able to better extract performance from both vault and bank parallelism. Nevertheless, such proposals suffer from inter-vault communication and TSV bandwidth limitation.

The most efficient way to gain performance with NDP when considering DRAM-based memories is to access data directly on the memory row buffers on each access (considering that such buffer will be filled by contiguous data). Should we be able to access all such data at once, we would theoretically able to explore all the internal bandwidth available in the memory. Thus, in order to offer the best performance possible, an NDP architecture must adjust to the underlying 3D-stacked memory to use as much of the bandwidth as possible. In considering a SIMD instruction approach such as VIMA, this means adjusting the width of vector operands according to the number of independent vaults and the size of their row buffers.

For instance, if we consider the HMC 2.1 (Hybrid Memory Cube Consortium, 2014), we have 32 independent vaults, each with a 256 B row buffer. Assuming parallel accesses to all 32 vaults, 8192 B are available on the row buffers per access. Since each vault in this configuration has 8 banks that can be accessed in a pipeline fashion, the device could possibly

Table 5.3: NDP vector size recommended for different 3D memory architectures.

3D-Stacked Memory	Number of Vaults	Row Buffer Size	Max. Number of Banks	Max. Request Size	NDP Vector Size
HMC 1.0	16	256 bytes	8	128 bytes	4096 bytes
HMC 2.1	32	256 bytes	16	256 bytes	8192 bytes
HBM	8	2 KBytes	16	128 bytes	16384 bytes
HBM2E	8	1 KByte	32	128 bytes	8192 bytes
HBM3	16	1 KByte	64	128 bytes	16384 bytes

provide 8192 B per access and thus, a NDP architecture could consider this size for its instruction operands in order to extract as much performance from the memory as possible. We could also expect that most of the latency to fetch the next chunk of 8192 B would be hidden by bank parallelism. It should be noted, however, that this line of thought does not necessarily translate to actual performance for every device as it ignores constraints such as internal transmission speed, maximum supported request sizes and the width of the connections between devices, as shall be discussed later in this chapter. We consider this situation for our experiments merely as a thought scenario to inform us on what could theoretically be the maximum performance we could achieve with VIMA when using each 3D-memory configuration if such perfect communication was feasible. Table 5.3 shows, for each memory configuration we are considering, its features that affect this aspect of our experiments and the vector size (last column) that would, in theory, most efficiently leverage both the internal bandwidth of the memory devices and the advantageous placement of a NDP architecture.

5.3 EXPERIMENT CHARACTERIZATION

For our experiments, we used 7 kernels as workload, each representing an algorithm commonly present in data-driven applications. *Memory Copy*, *Memory Set* and *Vector Sum* are data-streaming operations that cause a large portion of the data movement in typical consumer workloads (Boroumand et al., 2018) and in several Big Data applications (Cordeiro et al., 2021). *Selection* and *Projection* are common analytic database query operations that correspond to around 70% of the memory usage and processing time of database benchmark TPC-H (Kepe et al., 2019; Santos et al., 2022b). Lastly, two applications that feature data-reuse opportunities: *Stencil* represents a common behavior found in neural networks, image processing and CFD applications, and *Bloom Filter* is a ubiquitous data structure in string matching applications (Patgiri et al., 2018; Nayak and Patgiri, 2019; Sengupta and Rana, 2020). We used datasets of 8 MB, 16 MB, 32 MB and 64 MB for all workloads considered and obtained the application traces for simulation using Pin (Bach et al., 2010) tool. The dataset sizes were chosen to observe the impacts of the baseline cache and DRAM usage. All applications were coded and debugged using our Intrinsics-VIMA library using a common IDE and compiled using gcc version 7.5.0 with -O3 -static optimization flags.

Next we describe each application:

MemSet: traverses one vector setting every position to the same value.
MemCopy: copies all elements from vector to another vector starting at a different memory address.
VecSum: adds the elements of two vectors and stores the result in a third vector.
Selection: compares values in a vector with a filter value, storing a bitmap result.
Projection: loads elements from memory according to a bitmap mask and stores results in a

separate vector.
Stencil: computes a 5-point stencil convolution over a matrix and stores the result in an output matrix.
Bloom Filter: sets a bloom filter data structure with a set of items and uses another set to probe the structure for item membership.

Our experiments consider three distinct situations with increasing constraints regarding interconnection between VIMA and the memory:
Perfect: we assume the TSV connections in the 3D-stacked memory is able to provide all the data from the row buffers at a single cycle, considering a sustained pipeline of memory requests being issued to the 3D-stacked memory. Each request issued by VIMA is equal to row-buffer size and VIMA is able to receive this exact amount of data per cycle.
Max. Request Size: the 3D interconnection observes the maximum request size supported by each 3D-stacked memory design according to its specification. On each request VIMA issues is of that same size. VIMA issues as many requests as necessary to fetch each the full operand of an instruction according to this size limitation.
64 B Request Size: we assume the connection between VIMA and the main memory limits communication to the size of one line of the host processor cache per cycle (64 B). Each data request issued by VIMA to the memory is of this size (i.e. 64 B) and it issues as many requests as necessary to fetch each operand according to this limitation.

5.4 EXECUTION TIME RESULTS

We now present the execution time results for VIMA considering all three scenarios we described regarding the internal interconnection between DRAM row-buffers and VIMA. The width of the vectors considered by VIMA were adjusted according to each specific device as cited in Table 5.3. For brevity and clarity, we limit discussion in this section to the results to the memory configurations of the HMC 2.1 and HBM3. HMC 2.1 was chosen due to its very promising results in our execution time simulation experiments while HBM3 is considered for being the most recent JEDEC standard. Appendix B shows results considering all other aforementioned memory devices. To evaluate VIMA against the strongest possible baseline, all baseline experiments consider a HMC 2.1 main memory, as that memory organizations yielded the best performance for the x86 simulations.

5.4.1 Perfect interconnection and request size scenario

Figure 5.2 shows the results for all experiments considering a perfect interconnection and request size scenario. The *memory set* application is a clear example of a data streaming application, being composed of mainly one operation that stores an immediate value in each entry of a vector. As can be seen on Figure 5.2(a), the advantage VIMA has over the baseline shrinks as the input size grows. This happens due to the multithreaded nature of the baseline we are considering, as it suffers from the overhead of splitting the workload at the start of processing and aggregating all results when processing is finished. As input size grows, this overhead becomes a less significant portion of the overall execution time and thus the extent of the advantage of the NDP approach becomes more realistic. This applies to every application with primarily data streaming behavior.

Assuming the optimistic view in which the 3D-stacked memory device to which VIMA is attached is able to fully fill its row buffers with new data every cycle and a connection between memory and VIMA that allows for such data to be consumed at the same pace, the width of the vector considered by VIMA is the main determining factor for execution time performance when

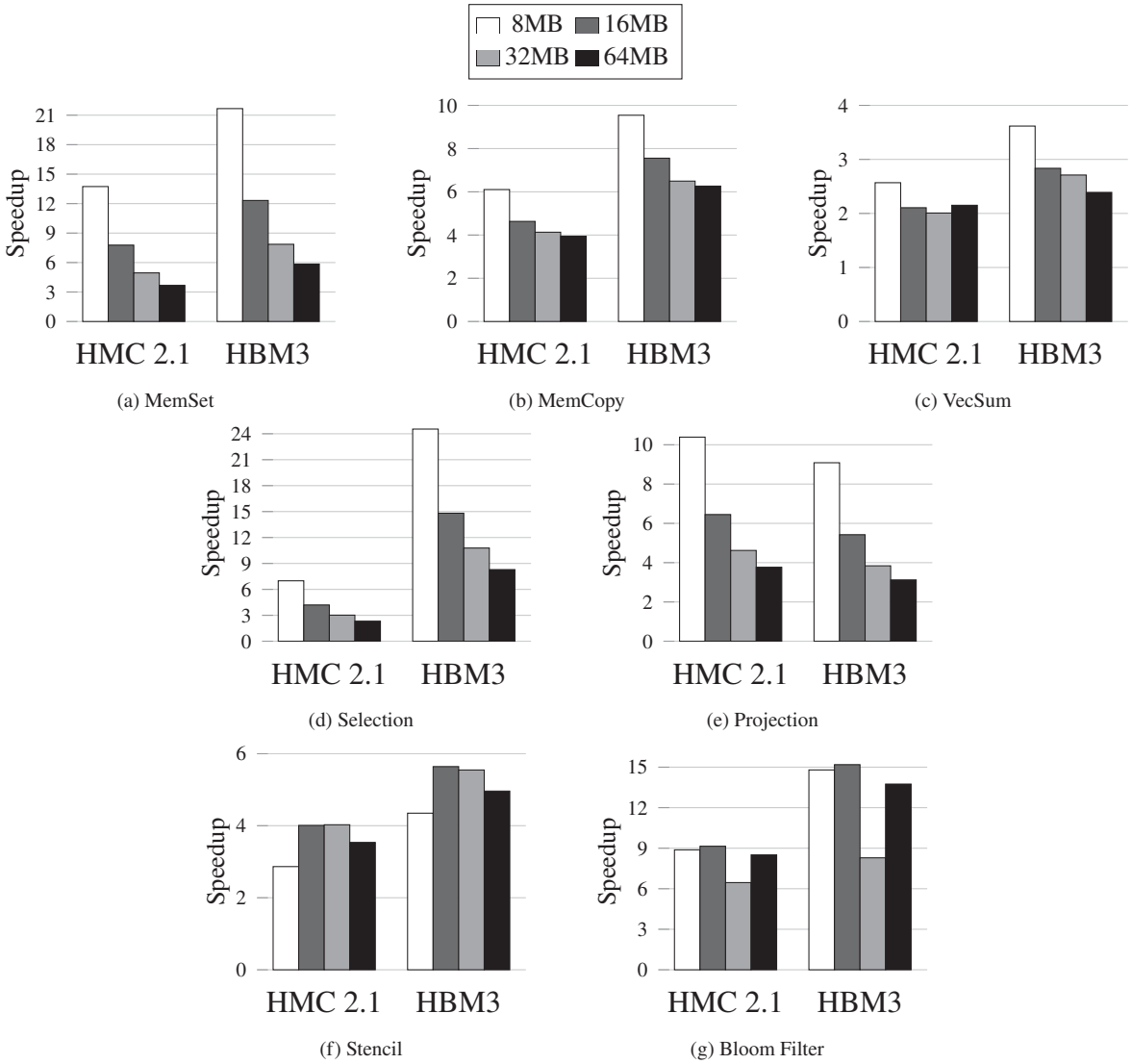


Figure 5.2: Execution time results of VIMA executing all workloads with perfect access to 3D-stacked memory row buffers normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

considering data streaming applications, as data throughput is the main limitation for processing speed for this class of application.

Thus, the memory configuration that would theoretically be able to offer the most throughput, HBM3, with which VIMA uses 16384 B vector operands, should see the largest speedup. In fact, as shown in Figure 5.2(a) in the experiment considering that design VIMA maintains a speedup of 5.84× over the baseline even at the largest input size considered.

For applications such as *memory copy* (Figure 5.2(b)) and *vector sum* (Figure 5.2(c)), VIMA sees a more modest performance improvement. This happens because these applications load two operands from the memory per instruction, as opposed to *memory set*, which only fetches one. Because of the load-ahead mechanism in VIMA, operands of multiple buffered VIMA instructions can be loaded in parallel, causing pressure applied to the memory to be kept high even for an application based on single operand instructions such as *memory set application*. However, this means that for applications that load multiple operands per instruction, such as *memory copy* and *vector sum*, although they apply pressure to the memory for a longer period, the rate at which data is provided does not become higher. As a result, while the baseline is able to

apply more pressure to the memory when executing these applications than it does with *memory set*, as each instruction causes two memory requests instead of only one, the same is not true for VIMA. This can be noticed in the experiment results with the less significant speedup. While the baseline now issues two data requests per instruction, VIMA is unable to apply more pressure to the memory because each single vector operand already takes up all the available bandwidth.

The *selection database query operator* and the *projection database query operator* results follow a similar pattern since both applications are also mainly based on data streaming behavior. For these applications VIMA has a slightly larger edge, up to 24× faster with the HBM-3 for selection and 15× for projection, because both applications are based on more complex operations than the ones used for the previous workloads. It is likely that this larger difference is caused by the longer execution latency of the instructions on the baseline and also by limitations of the simulation environment in simulating such instructions.

With the *stencil* and *bloom filter applications*, the cache memories of both the baseline and VIMA begin to impact results, since both applications have some level of reuse behavior and benefit from the cache hierarchies. While *Stencil* is composed of a very simple algorithm with clear data reuse within its main loop, it still behaves mostly as a data streaming application in that once its main loop goes past any data element, that element is never accessed again. The *bloom filter application*, on the other hand, has a different behavior in that it accesses the same data structure repeatedly throughout the entire execution and its results on Figure 5.2(g) reflect this. While the baseline benefits from its cache hierarchy, VIMA benefits as well from the fact that the data structures used in the application are small enough to fit in its much smaller cache. Consequently, VIMA achieves a speedup of up to 15× when considering the experiment the HBM-3 memory. Although *stencil* also reuses data that is found in the dedicated cache memory, this data gets reused much fewer times, causing the speedup for that application to be much more modest than the one achieved for the *bloom filter* application.

The speedup results we were able to obtain considering this optimistic configuration display the potential for improvement in execution time performance of a NDP architecture. We must keep in mind, however, that these experiments significantly relax constraints that are present even when considering the privileged placement of a NDP architecture relative to the main memory. While speedups of over 20× against a modern 16-threaded traditional architecture using a single-core system are enticing, to have a better understanding of the results we may reasonably expect from this type of approach, we must consider a number of constraints we have chosen to ignore up until now to investigate what optimal performance could look like.

5.4.2 Maximum specified request and interconnection size scenario

Figure 5.3 shows the results for all experiments considering the maximum specified request and interconnection size scenario.

The first constraints must be considered to obtain a more realistic estimate of the performance improvements we can expect from VIMA is the maximum request size each memory configuration supports. While considering our optimistic experiments, we assumed each 3D-stacked memory device was capable of fully transmit the entire row buffer to the the VIMA cache within a single cycle, thus also replenishing the entire row buffer of each vault with new data every cycle, thus also assuming that every request sent to each vault was the same size as the width of its row buffer. Unfortunately, that seems unfeasible and is reflected in the specification of the devices regarding the maximum request size they support. This can be viewed as a proxy for what data transmission rates the TSV interconnections are able to handle. For instance, the HBM configurations support a maximum request size of 128 B, although their row buffer widths range from 1 KB to 2 KB. Table 5.3 shows the maximum request size each 3D-stacked memory

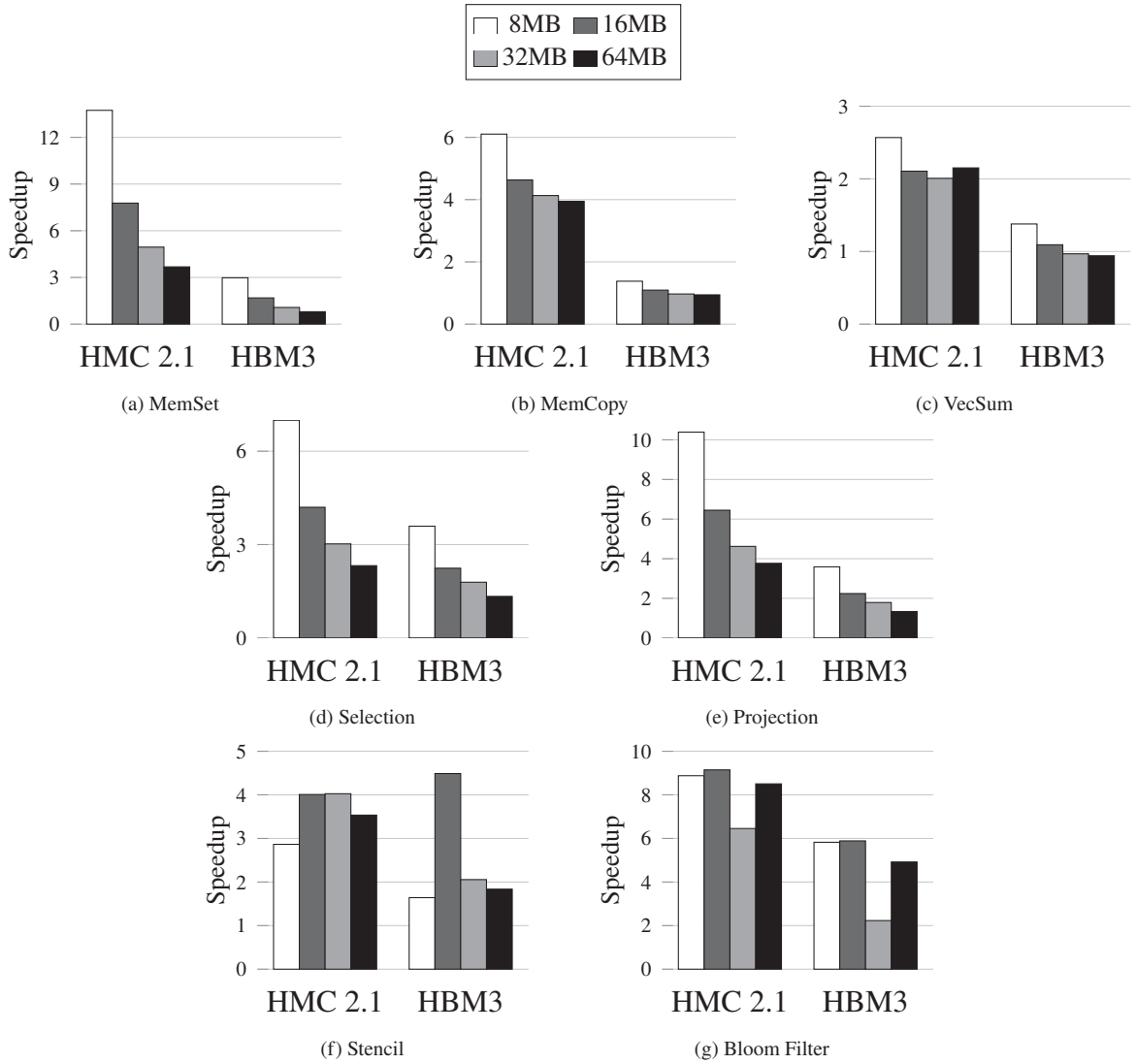


Figure 5.3: Execution time results of VIMA executing all workloads with maximum request size supported by each 3D-memory device normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

considered in our experiments. Figure 5.3 shows the simulation results for the seven applications we are using as workload for the two memory configurations considering the maximum request size of each configuration. Results for all memory configurations cited in Table 5.3 can be found on Appendix B.

It is immediately clear that the results for the HMC 2.1 device are superior for almost every application and input size simulated. This is because the maximum request size of this 3D-stacked memory, as can be seen on Table 5.3 is equal to the size of its row buffers, which means it is theoretically able to support functioning even at the optimistic situation we considered in our previous experiments.

On the other hand, experiments with the HBM designs achieve much more modest results. Such effect is due to the HBM devices having fewer independent vaults than the Hybrid Memory Cube and a row buffer width that is much larger than its maximum request size. This causes VIMA to take 8 to 16 cycles/requests to consume the data in each row buffer, thus being unable to efficiently utilize the bandwidth of the device as it is more geared toward bank

parallelism than vault parallelism. Naturally, in practice, this request size limitation is likely set considering the data transmission rate the hardware is able to sustain.

5.4.3 64 B interconnection and request size scenario

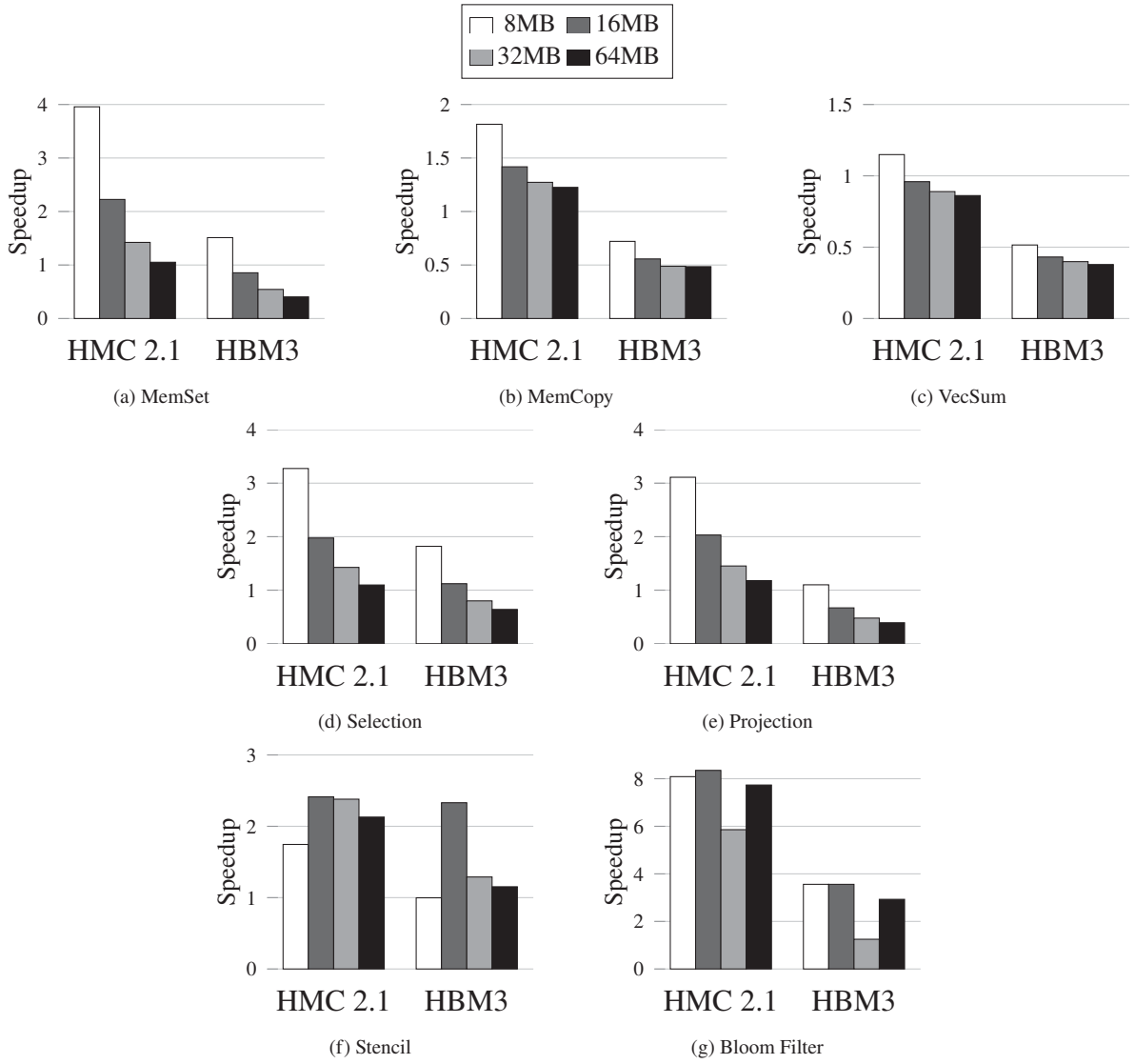


Figure 5.4: Execution time results of VIMA executing all workloads with with 64 B request size normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

Figure 5.4 shows the results for all experiments considering a 64 B interconnection and request size scenario. The second constraint we must consider to obtain a realistic estimate of the performance we can expect from VIMA regards the number of connections between main memory and near-data processing device. In our first experiment, we assumed communication between the main memory and VIMA could handle transferring up to 16384 B per cycle, which would require 131072 individual connections between the circuits. This is unfeasible and would have a very high area and energy consumption cost. The conditions we simulated on our second set of experiments have a much smaller requirement, needing only a fraction of those connections to handle 128- and 256-byte requests.

Still, a very reasonable assumption is that the connection between the devices is the same size as the cache line considered by the host processor in its cache hierarchy, since that is

likely the size assumed by hardware manufacturers to be most commonly used. This makes for a much more realistic design and still allows for good near-data performance, even if it limits significantly the data throughput we can expect to be able to extract from the memory. Figures5.4 shows the simulation results for the seven applications we are using as workload for each of memory configurations considering the 64 B request size.

As seen in the simulation results, even considering much harsher constraints, VIMA is still able to either match or outperform a 16-thread x86 baseline in most situations, especially with the HMC 2.1 memory device. This means, for instance, that VIMA could eliminate the need for 15 cores, freeing them for other tasks.

5.5 ENERGY CONSUMPTION RESULTS

We now discuss results regarding the energy savings of VIMA in relation to the x86 baseline. Energy consumption was estimated using CACTI and Multicore Power, Area, and Timing (McPAT) tools, as is commonly done in related work. Both tools were used to evaluate power, area, and timing parameters according to circuitry as modeled (Li et al., 2009). Figure 5.5 shows our experiment results.

We see in our experiment results that VIMA is largely more energy-efficient than the 16-thread baseline for almost all experiments, regardless of application, underlying 3D-stacked memory configuration or input size. All results here consider the maximum request size supported by each memory device and show a minimum reduction in energy consumption of 51% across all applications and input sizes.

The results for the *bloom filter application* are consistently the most positive across all input sizes and memory configurations, but they reveal a limitation of our simulation environment. The algorithm used by that workload includes a number of instructions that would be impractical to implement on hardware at the vector width required, as they would need a very large number of multiplexers and connections. This would greatly increase requirements of area, consequently also significantly increasing energy consumption of the entire design. Our simulation environment considers an ISA that does not include those highly complex instructions, such as memory gather and memory scatter. Simulation results for this application would likely differ significantly from these in a simulation environment that implements these instructions.

The energy efficiency VIMA offers is due mainly because of its usage of a single processing core as host, coupled with a more efficient use of the main memory. While energy consumption with processing cores for the baseline ranges from 42% to 81% of the total consumption, processing core consumption for VIMA never accounts for more than 30% of total system consumption, regardless of application, underlying memory device, input size or assumption regarding data access efficiency.

5.6 DATA THROUGHPUT RESULTS

Figure 5.6 shows results for average throughput for our experiments considering all data streaming kernels.

The graph considers absolute values of average data throughput achieved by VIMA considering maximum request sizes and also a x86 with a number of cores varying between 1 and 16. All results refer to the largest input size considered in our experiments, 64 MB.

These results show how better usage of available data throughput is the main reason why VIMA performs better than a traditional architecture when running data streaming applications. It is also clear that whenever VIMA is unable to surpass the performance of the baseline

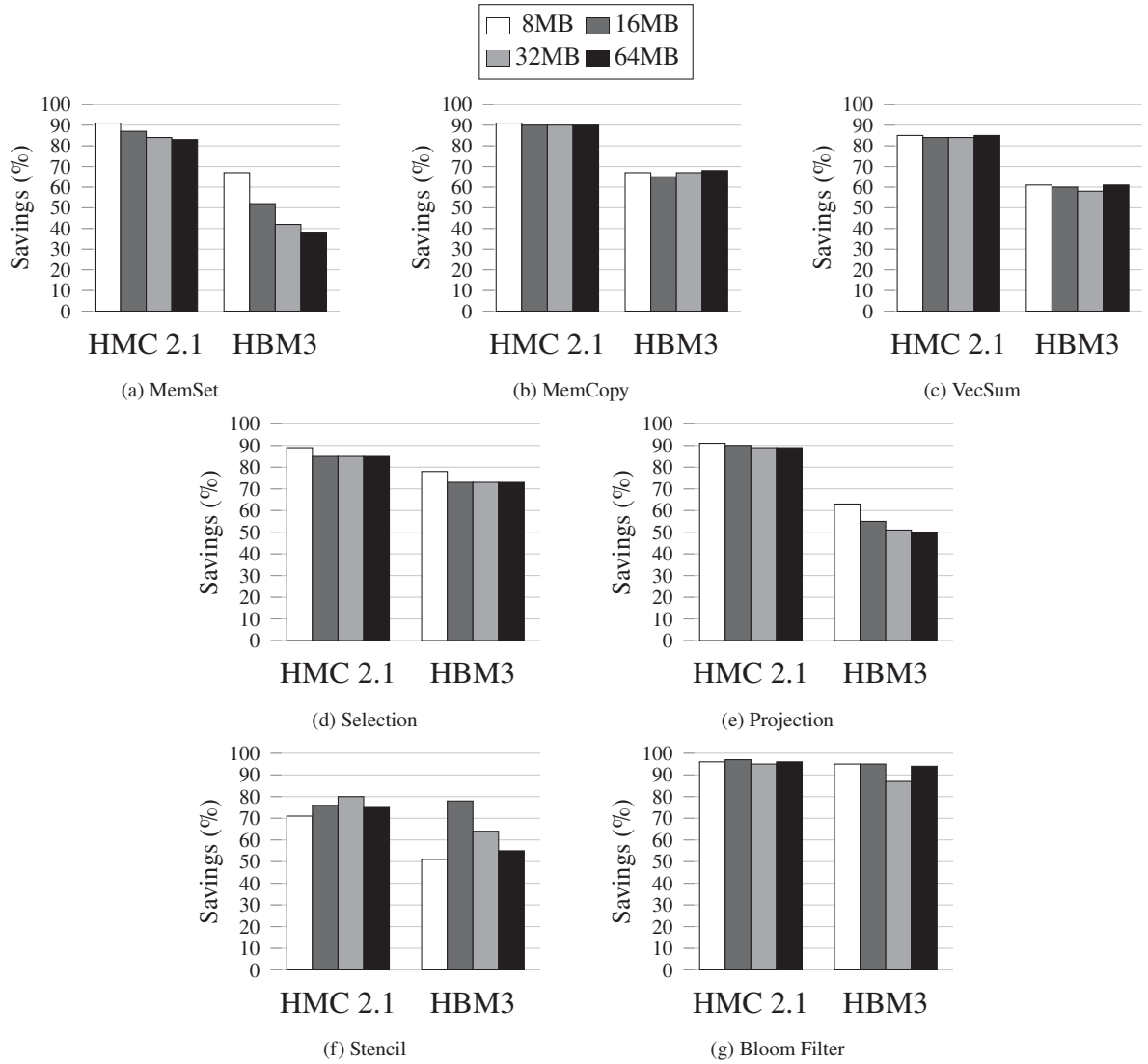


Figure 5.5: Energy savings results of VIMA executing all workloads with maximum request size supported by each 3D-memory device normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

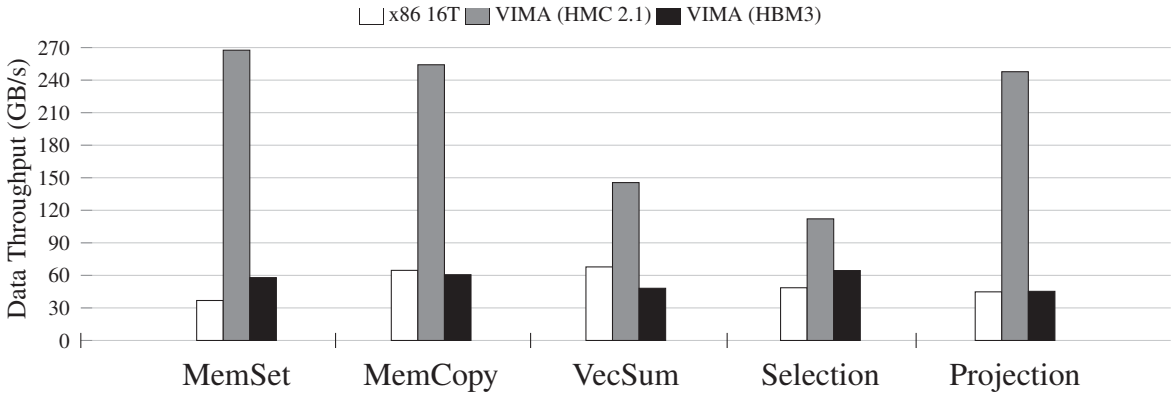


Figure 5.6: Data throughput results executing all data streaming applications. Absolute values in GB/s.

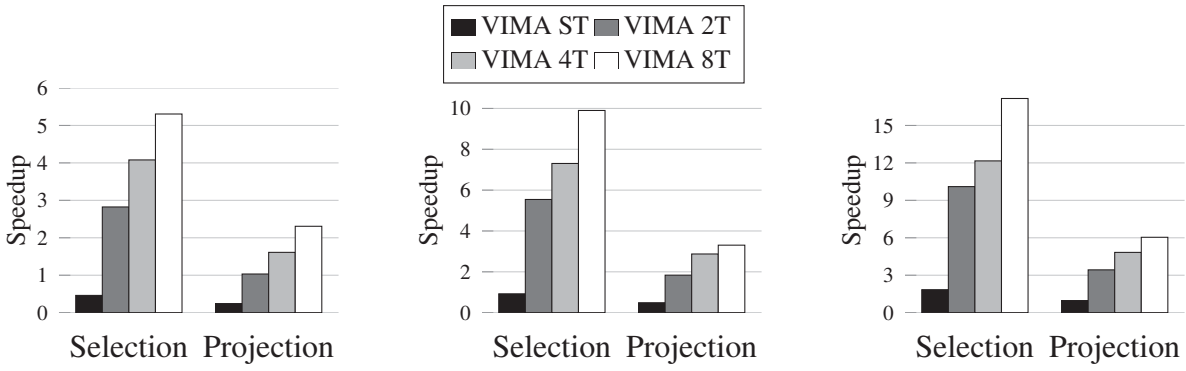
regarding data throughput, it fails to surpass execution time performance as well, as can be seen in Figures 5.3(b) and 5.3(c) for the HBM3 results.

VIMA achieves a data throughput of 267 GB/s when executing *memset application* on the HMC 2.1, which is both very high and well within the theoretical maximum specified in the specification of the HMC device (Hybrid Memory Cube Consortium, 2014), 320 GB/s. On the other hand, the best throughput we were able to achieve with VIMA with the HBM3 memory was 64 GB/s, when running the *selection database query*, although the theoretical maximum bandwidth of the HBM3 device is 512 GB/s (Association, 2022).

This discrepancy in results likely happens because of how the two devices are organized. While the HMC 2.1 has opportunities for vault parallelism, being split in 32 independent vaults, the HBM3 is split only in 16. The fact that the maximum request size supported by the HMC is equal to the width of its row buffers also causes VIMA to achieve better performance with this design, as is it able to better utilize also the bank parallelism available in the device. In contrast, although the HBM has much larger number of banks per vault, 64 against 8 in each HMC vault, since its maximum request size is 128 B with a 1 KB-wide row buffer, VIMA takes much longer to consume the data that is placed on the row buffer. This causes the device to be unable to efficiently utilize the bank parallelism available as banks in the vaults become ready for new accesses long before VIMA is ready to access them.

5.7 MULTITHREADING RESULTS

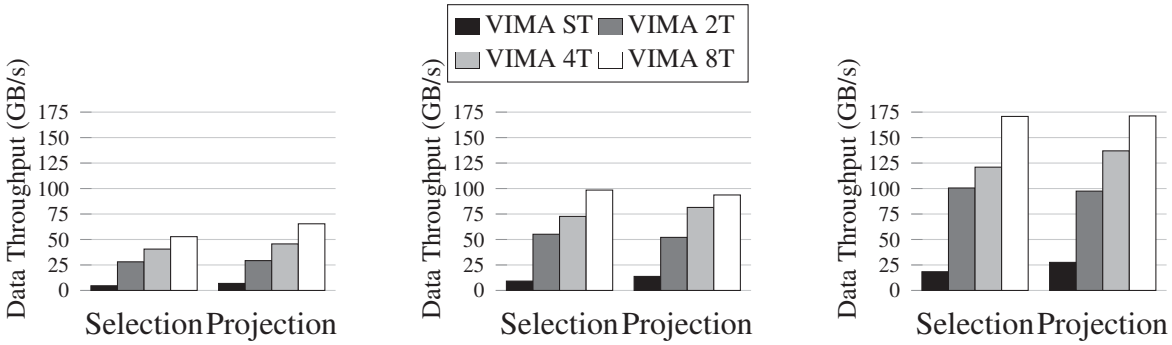
We ran experiments to analyze the performance of a multithreaded system using VIMA and the speedup and data throughput results can be seen on Figures 5.7 and 5.8. Our experiments considered the selection and projection database query operators running on a VIMA-enabled system with increasing vector widths (256 B, 512 B and 1024 B) and number of cores (1, 2, 4 and 8 cores). The underlying memory chip we used was the HMC 2.1, as it has generally shown the most advantageous results so far, and consider the 64 MB input size for both workloads.



(a) 256 B Vector (b) 512 B Vector (c) 1024 B Vector

Figure 5.7: Speedup of VIMA over baseline running the selection and projection database queries with a varying number of processing threads and (a) 256 B vector operands, (b) 512 B vector operands and (c) 1024 B vector operands. Values higher than 1 indicate improvement in performance over the baseline.

The speedup results, which are normalized to a 16-thread x86 system, show that when using smaller vector operands, VIMA is unable to match the baseline performance when running on a single-threaded system. However, even with only one additional core, it outperforms the baseline for the selection workload and matches baseline performance for projection. This advantage scales with larger vectors and a higher number of threads, achieving a 17× improvement in execution time over the baseline for the selection database query workload at 8 threads with a 1024 B vector operand width.



(a) 256 B Vector (b) 512 B Vector (c) 1024 B Vector

Figure 5.8: Data throughput of VIMA running the selection and projection database queries with a varying number of processing threads and (a) 256 B vector operands, (b) 512 B vector operands and (c) 1024 B vector operands. Values higher than 1 indicate improvement in performance over the baseline.

The data throughput results, as seen in Figure 5.8, show why there is such an improvement in execution time performance with the addition of extra cores. As is immediately clear when analyzing the results, the single-thread system with the smaller VIMA vectors is unable to apply enough pressure to the memory, thus failing to achieve very high data throughput. Although the load-ahead mechanism helps VIMA extract more throughput from the memory, not enough instructions are ever in the VIMA instruction buffer at the same time to allow it to exploit the full potential of the vault parallelism present in the HMC 2.1 memory chip. However, with the addition of extra cores, many more instructions are issued at the same time, which then enables VIMA, through the load-ahead mechanism, to load the operands of more instructions out-of-order, thus utilizing much more of the bandwidth the memory device is able to offer.

As a result, even with a 256 B vector, VIMA outperforms the baseline by over 2× even with only 2 cores at the selection database query. Under the same conditions it at least matches performance of the baseline for the projection workload, achieving a 2× speedup when using 8 cores. This trend remains true for the results of the experiments with 512 B and 1024 B operands.

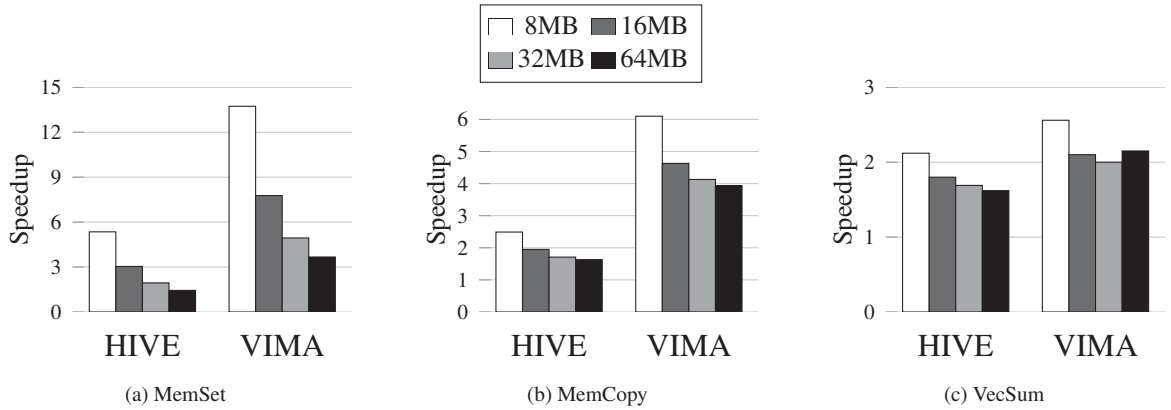
5.8 STATE-OF-THE-ART COMPARISON

To compare VIMA with another NDP architecture that falls under the same category, we ran experiments using HIVE (Alves et al., 2016). HIVE is a similar architecture to VIMA in that it also leverages the high parallelism of a 3D-stacked memory, namely the HMC 2.0. It also mirrors VIMA in that its processing is based on FUs and offloading is done through large NDP-specific SIMD instructions that are added to the processor ISA as extensions.

The two architectures differ in how they store data and handle instructions, which has significant consequences for the performance they can achieve. For instance, HIVE uses a register bank for dedicated data storage, which causes it to require a locking and unlocking mechanism to manage access to its resources. Another important difference is that, in order to preserve precise exceptions, we assume in our experiments that HIVE only handles one instruction at a time, meaning all processing and data loading is done strictly in program order. The behavior of its original design does not maintain precise exceptions and does not include this limitation.

We considered the *MemSet*, *MemCopy* and *VecSum* workloads with 8 MB, 16 MB, 32 MB, 64 MB input dataset sizes on a single-core system. The other workloads we considered for VIMA were either not available for HIVE or include instructions not implemented in the HIVE ISA. Thus, operand size for instructions of both architectures was kept the same, as was

the underlying 3D-stacked memory. All results are normalized to a 16-thread baseline with AVX-512 capabilities. Figure 5.9 shows the results of our experiments.



(a) MemSet (b) MemCopy (c) VecSum

Figure 5.9: Speedup of VIMA and HIVE over 16-thread baseline running (a) *memory set application*, (b) *memory copy application* and (c) *vector sum application*. Values higher than 1 indicate improvement in performance over the baseline.

As the results show, VIMA has a distinct advantage over HIVE, mainly because of its contributions to the state-of-the-art of this class of NDP device. While HIVE still out performs the 16-thread baseline for all workloads, the execution time performance VIMA is able to provide is superior. The memory set application results clearly highlight the advantage caused by the load-ahead mechanism implemented by VIMA: although both devices function with the same vector operand width and same underlying memory chip, VIMA is able to extract much more performance, outperforming HIVE by over 2.5× even at the largest input size considered. Results for memory copy follow a similar trend, with VIMA providing a 2.4× improvement in execution time performance over HIVE. While improvements for the vector sum application are not as significant, for the reasons already discussed in Section 5.4, VIMA still outperforms HIVE by at least 32% at the largest input size.

5.9 SUMMARY

In this chapter, we evaluated the performance of VIMA, our NDP architecture proposal, under many possible conditions. We simulate several Big Data kernels and several possible configurations for the underlying main memory architecture.

Our first set of experiments was purely theoretical and assumed an optimistic situation. In this situation the main memory of the system is capable to transmit its row buffers in a single cycle to VIMA, thus being able to take advantage of as much data access parallelism as the memory could provide. Our simulations found that, under such circumstances, VIMA would be able to outperform a modern 16-thread x86 baseline by up to 10× regarding execution time when running data streaming applications, while reducing energy consumption by 85%.

On a more realistic set of experiments considering the largest request size supported by each distinct 3D-stacked memory configuration, results also showed significant improvements. While the improvement in execution time reached a maximum of 3.96× for large datasets, VIMA was still able to take advantage of a large portion of the bandwidth available in the HMC 2.1 memory chip, achieving a throughput of up to 267GB/s and a reduction of at least 75% in energy consumption across all workloads. This set of experiments also revealed a clear advantage of the HMC memory chip over the HBM design for our VIMA architecture due to the difference in row buffer size and vault parallelism between the two architectures.

Finally, simulating a more limited configuration in which VIMA is only able to receive 64 B per cycle, results were still positive. VIMA provided a 2.12× improvement in execution time for data streaming applications, achieving a data throughput of 76GB/s and a minimum reduction in energy consumption of 52%.

When considering multithreading, a feature made possible by the contributions of VIMA to fine-grain FU-based NDP accelerators, our simulations showed that VIMA was able to deliver good performance even at smaller vector operand widths. We considered a 256 B vector operand width with an underlying HMC 2.1 memory and were able to achieve a speedup of up to 5× when running common database query operators on VIMA with a combination of our load-ahead mechanism and instructions issued from eight parallel threads.

Lastly, we compared our proposal with HIVE, a state-of-the-art NDP architecture that follows the same general principles as VIMA. Our results showed that, due to the improved efficiency of VIMA, it is able to reduce execution time compared to HIVE by over 2× with the same underlying memory chip and vector length.

Portions of the work presented on this chapter has been published at Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) (Cordeiro et al., 2021; Santos et al., 2022b), International Symposium on Performance Analysis of Systems and Software (ISPASS) (Santos et al., 2022a), and XXII Escola Regional de Alto Desempenho da Região Sul (ERAD/RS) (Santos and Alves, 2022). A research paper featuring portions of this work is currently under review at The Journal of Supercomputing.

6 CONCLUSIONS AND FUTURE WORK

With the rise of data-driven applications, the memory wall continues to be a significant bottleneck. Near-Data Processing (NDP) attempts to alleviate this issue by placing processing near the data, thus avoiding most of the data movement that would be otherwise be necessary as most required data would be fetched from the main memory, causing high latencies and energy consumption. With the advent of 3D-stacked memories, NDP research has seen a wealth of new architecture proposals that take advantage of the efficient logic-storage integration such devices offer to implement logic processing near the data.

In this thesis we propose Vector-In-Memory Architecture (VIMA), a NDP architecture that improves on the existing state-of-the-art. By expanding a common Functional Unit (FU)-based processing model found in fine-grain NDP architecture proposals in the literature, VIMA implements precise exceptions near the data and enables efficient usage of internal bandwidth in the memory. Our design also leverages such advancements to enable near-data multithreading.

The experimental results we obtained show that a single-thread VIMA-enabled processing can successfully extract enough data throughput from the memory to outperform a 16-thread x86 baseline with traditional Single Instruction Multiple Data (SIMD) extensions. While providing a speedup of at least 2× when dealing with data-streaming applications, it also reduces energy consumption by as much as 90%, mostly due to its function with a single processing core. Against a closely related existing state-of-the-art NDP architecture, VIMA is able to speed up execution between 32% and 153% due to its improved usage of data throughput. VIMA is hardware-agnostic and can function with either High Bandwidth Memory (HBM) or Hybrid Memory Cube (HMC), the two most common 3D-stacked memory products commercially available. It does, however, achieve its best results with the HMC 2.1 design.

At the beginning of this thesis, we posed the following hypothesis: **it is possible to provide precise exceptions, improved memory access and multithreading with fine-grain NDP to speedup data streaming applications**. As discussed in chapters 4 and 5, our proposal implements precise exceptions while optimizing usage of the underlying memory. It indeed achieves improved performance both against a modern multithreaded x86 system and a state-of-the-art fine-grain NDP system when processing several common data streaming applications, which confirms our hypothesis.

6.1 FUTURE WORK

Our results with the HMC 2.1 memory chip have been very positive, but the HBM has become the Joint Electron Device Engineering Council (JEDEC) standard, causing development of the HMC to be discontinued. One avenue for future work is to explore how we can modify VIMA to better explore the throughput available in a HBM memory. On the other hand, if we consider the results of VIMA with a HBM3 against a baseline also using a HBM3 memory, VIMA is still able to provide a significant execution time improvement. Such results can be seen on Appendix B.

The approach we use to integrate VIMA with the host system requires modifications to the processor. An interesting opportunity for future work is to explore options to make integration more seamless so as to facilitate widespread adoption by removing this barrier. Some existing work in the literature has already proposed a few strategies to achieve this with other NDP proposals (Santos et al., 2021a).

Other avenues for future work include implementing a NDP-aware cache coherence mechanism such as CoNDA (Boroumand et al., 2019) or LazyPIM (Boroumand et al., 2016); and exploring the possibility of a system with multiple instances of VIMA in a Non-Uniform Memory Access (NUMA) architecture.

6.2 LIST OF PUBLICATIONS

The list below presents the published works related to this thesis proposal, ordered by publication year:

- CORDEIRO, ALINE S.; **SANTOS, SAIRO R.**; MOREIRA, FRANCIS B.; SANTOS, PAULO C.; CARRO, LUIGI; ALVES, MARCO A. Z. Machine Learning Migration for Efficient Near-Data Processing In: 29th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2021.

- SANTOS, PAULO C.; MOREIRA, FRANCIS B.; CORDEIRO, ALINE S.; **SANTOS, SAIRO R.**; KEPE, TIAGO R.; CARRO, LUIGI; ALVES, MARCO A. Z. Survey on Near-Data Processing: Applications and Architectures In: Journal of Integrated Circuits and Systems, 2021.

- CORDEIRO, ALINE S.; **SANTOS, SAIRO R.**; MOREIRA, FRANCIS B.; SANTOS, PAULO C.; ALVES, MARCO A. Z. Efficient Machine Learning execution with Near-Data Processing In: Microprocessors and Microsystems, 2022.

- **SANTOS, SAIRO R.**; MOREIRA, FRANCIS B.; KEPE, TIAGO R.; SANTOS, PAULO C.; ALVES, MARCO A. Z. Advancing Database System Operators with Near-Data Processing In: 2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), 2022.

- **SANTOS, SAIRO R.**; ALVES, MARCO A. Z.; CORDEIRO, ALINE S.; MOREIRA, FRANCIS B.; SANTOS, PAULO C.; CARRO, LUIGI Vector In Memory Architecture for simple and high efficiency computing In *arXiv preprint arXiv:2203.14882*.

- **SANTOS, SAIRO R.**; ALVES, MARCO A. Z. Impacto da Largura do Vetor de Instruções SIMD em Arquiteturas de Processamento Próximo à Memória In XXII Escola Regional de Alto Desempenho da Região Sul (ERAD/RS), 2022.

- **SANTOS, SAIRO R.**; KEPE, TIAGO R.; MOREIRA, FRANCIS B.; ALVES, MARCO A. Z. Advancing Near-Data Processing with Precise Exceptions and Efficient Data Fetching In: 2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2022.

We also have the follow paper under review:

- **SANTOS, SAIRO R.**; MOREIRA, FRANCIS B.; KEPE, TIAGO R.; SANTOS, PAULO C.; ALVES, MARCO A. Z. Advancing Database System Operators with Near-Data Processing In: The Journal of Supercomputing, 2022.

REFERENCES

Aga, S., Jeloka, S., Subramaniyan, A., Narayanasamy, S., Blaauw, D., and Das, R. (2017). Compute caches. In *Int. Symp. on High Performance Computer Architecture (HPCA)*.

Ahmed, H., Santos, P. C., Lima, J. P., Moura, R. F., Alves, M. A., Beck, A. C., and Carro, L. (2019). A compiler for automatic selection of suitable processing-in-memory instructions. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

Ahmed, I., Zoranic, A., Javaid, S., Richard, G., and Roussev, V. (2013). Rule-based integrity checking of interrupt descriptor tables in cloud environments. In *IFIP International Conference on Digital Forensics*, pages 305–328. Springer.

Ahn, J. et al. (2015a). A scalable processing-in-memory accelerator for parallel graph processing. In *Int. Symp. on Computer Architecture*.

Ahn, J., Yoo, S., Mutlu, O., and Choi, K. (2015b). Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 336–348. IEEE.

Ali, M. F., Jaiswal, A., and Roy, K. (2020). In-memory low-cost bit-serial addition using commodity dram technology. *Trans. on Circuits and Systems (TCS)*.

Alian, M. et al. (2018). Application-transparent near-memory processing architecture with memory channel network. In *Int. Symp. on Microarchitecture (MICRO)*.

Alves, M. A., Santos, P. C., Diener, M., and Carro, L. (2015a). Opportunities and challenges of performing vector operations inside the dram. In *Int. Symp. on Memory Systems (MEMSYS)*.

Alves, M. A., Santos, P. C., Moreira, F. B., Diener, M., and Carro, L. (2015b). Saving memory movements through vector processing in the dram. In *Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*.

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

Alves, M. A. Z. et al. (2015c). Sinuca: A validated micro-architecture simulator. In *Int. Conf. on High Performance Computing and Communications*.

Alves, M. A. Z. et al. (2016). Large vector extensions inside the hmc. In *Design, Automation & Test in Europe Conf.*

Alves, M. A. Z., Santos, S., Cordeiro, A. S., Moreira, F. B., Santos, P. C., and Carro, L. (2022). Vector in memory architecture for simple and high efficiency computing.

AMD (2015). High bandwidth memory.

Ando, K., Ueyoshi, K., Orimo, K., Yonekawa, H., Sato, S., Nakahara, H., Takamaeda-Yamazaki, S., Ikebe, M., Asai, T., Kuroda, T., et al. (2017). Brein memory: A single-chip binary/ternary reconfigurable in-memory deep neural network accelerator achieving 1.4 tops at 0.6 w. *Journal of Solid-State Circuits*.

Angizi, S. et al. (2020). Pim-assembler: A processing-in-memory platform for genome assembly. In *Design Automation Conf. (DAC)*.

Angizi, S., Sun, J., Zhang, W., and Fan, D. (2019). Aligns: A processing-in-memory accelerator for dna short read alignment leveraging sot-mram. In *Design Automation Conf. (DAC)*.

Association, J. S. S. T. (2013). High bandwidth memory (hbm) dram, specification jesd235. `https://www.jedec.org/`.

Association, J. S. S. T. (2022). High bandwidth memory (hbm) dram, specification jesd238. `https://www.jedec.org/`.

Awan, A. J., Brorsson, M., Vlassov, V., and Ayguade, E. (2016). Micro-architectural characterization of apache spark on batch and stream processing workloads. In *Int. Conf. on Big Data and Cloud Computing (BDCloud)*.

Awan, A. J., Ohara, M., Ayguadé, E., Ishizaki, K., Brorsson, M., and Vlassov, V. (2017). Identifying the potential of near data processing for apache spark. In *Int. Symp. on Memory Systems (MEMSYS)*.

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

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. et al. (2014). Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34.

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*.

Boroumand, A., Ghose, S., Patel, M., Hassan, H., Lucia, B., Ausavarungnirun, R., Hsieh, K., Hajinazar, N., Malladi, K. T., Zheng, H., et al. (2019). Conda: Efficient cache coherence support for near-data accelerators. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 629–642.

Boroumand, A., Ghose, S., Patel, M., Hassan, H., Lucia, B., Hsieh, K., Malladi, K. T., Zheng, H., and Mutlu, O. (2016). Lazypim: An efficient cache coherence mechanism for processing-in-memory. *IEEE Computer Architecture Letters*, 16(1):46–50.

Cadambi, S., Majumdar, A., Becchi, M., Chakradhar, S., and Graf, H. P. (2010). A programmable parallel accelerator for learning and classification. In *Int. Conf. on Parallel architectures and Compilation Techniques (PACT)*.

Cali, D. S., Kalsi, G. S., Bingöl, Z., Firtina, C., Subramanian, L., Kim, J. S., Ausavarungnirun, R., Alser, M., Gomez-Luna, J., Boroumand, A., et al. (2020). Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis. In *Int. Symp. on Microarchitecture (MICRO)*.

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

Cheng, M., Xia, L., Zhu, Z., Cai, Y., Xie, Y., Wang, Y., and Yang, H. (2017). Time: A training-in-memory architecture for memristor-based deep neural networks. In *Design Automation Conf. (DAC)*.

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

Cho, B. Y., Kwon, Y., Lym, S., and Erez, M. (2020). Near data acceleration with concurrent host access. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 818–831. IEEE.

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

Cordeiro, A. S. (2020). Porting machine learning algorithms to vector-in-memory architecture. Master's thesis, Universidade Federal do Paraná.

Cordeiro, A. S. et al. (2021). Machine learning migration for efficient near-data processing. In *Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*.

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

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. (DAC)*.

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. (DAC)*.

Devaux, F. (2019). The true processing in memory accelerator. In *Hot Chips Symp.*

Drebes, A., Chelini, L., Zinenko, O., Cohen, A., Corporaal, H., Grosser, T., Vadivel, K., and Vasilache, N. (2020). Tc-cim: Empowering tensor comprehensions for computing-in-memory. In *Int. Workshop on Polyhedral Compilation Techniques (IMPACT)*.

Drumond, M., Daglis, A., Mirzadeh, N., Ustiugov, D., Picorel, J., Falsafi, B., Grot, B., and Pnevmatikatos, D. N. (2017). The mondrian data engine. In *Int. Symp. on Computer Architecture (ISCA)*.

Drumond, M. et al. (2018). Algorithm/architecture co-design for near-memory processing. *Operating Systems Review*.

Eckert, C., Wang, X., Wang, J., Subramaniyan, A., Iyer, R., Sylvester, D., Blaaauw, D., and Das, R. (2018). Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *Int. Symp. on Computer Architecture (ISCA)*.

Eckert, Y., Jayasena, N., and Loh, G. H. (2014). Thermal feasibility of die-stacked processing in memory. In *MEMSYS '16: Proceedings of the Second International Symposium on Memory Systems*. Citeseer.

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*.

et al., V. S. (2017). Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Int. Symp. on Microarchitecture (MICRO)*.

Farmahini-Farahani, A., Ahn, J. H., Morrow, K., and Kim, N. S. (2014). Drama: An architecture for accelerated processing near memory. *Computer Architecture Letters (CAL)*.

Farmahini-Farahani, A., Ahn, J. H., Morrow, K., and Kim, N. S. (2015). Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *Int. Symp. on High Performance Computer Architecture (HPCA)*.

Gao, F., Tziantzioulis, G., and Wentzlaff, D. (2019). Computedram: In-memory compute using off-the-shelf drams. In *Int. Symp. on Microarchitecture (MICRO)*.

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

Gao, M. and Kozyrakis, C. (2016). Hrl: Efficient and flexible reconfigurable logic for near-data processing. In *Int. Smyp. on High Performance Computer Architecture (HPCA)*.

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

Ghiasi, N. M., Park, J., Mustafa, H., Kim, J., Olgun, A., Gollwitzer, A., Cali, D. S., Firtina, C., Mao, H., Alserr, N. A., et al. (2022). Genstore: A high-performance and energy-efficient in-storage computing system for genome sequence analysis. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Gupta, S. et al. (2019). Rapid: A reram processing in-memory architecture for dna sequence alignment. In *Int. Symp. on Low Power Electronics and Design (ISLPED)*.

Gupta, S., Imani, M., and Rosing, T. (2018). Felix: Fast and energy-efficient logic in memory. In *Int. Conf. on Computer-Aided Design (ICCAD)*.

Gómez-Luna, J., Hajj, I. E., Fernandez, I., Giannoula, C., Oliveira, G. F., and Mutlu, O. (2021). Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture.

Hadidi, R., Nai, L., Kim, H., and Kim, H. (2017). Cairo: A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory. *Trans. on Architecture and Code Optimization (TACO)*.

Haj-Ali, A., Ben-Hur, R., Wald, N., Ronen, R., and Kvatinsky, S. (2018). Not in name alone: A memristive memory processing unit for real in-memory processing. *IEEE Micro*.

Hajinazar, N., Oliveira, G. F., Gregorio, S., Ferreira, J. D., Ghiasi, N. M., Patel, M., Alser, M., Ghose, S., Gómez-Luna, J., and Mutlu, O. (2021). Simdram: a framework for bit-serial simd processing using dram. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

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

Hong, B., Kim, G., Ahn, J. H., Kwon, Y., Kim, H., and Kim, J. (2016). Accelerating linked-list traversal through near-data processing. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 113–124.

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].

Hsieh, K., Ebrahimi, E., Kim, G., Chatterjee, N., O'Connor, M., Vijaykumar, N., Mutlu, O., and Keckler, S. W. (2016a). Transparent offloading and mapping (tom) enabling programmer-transparent near-data processing in gpu systems. *ACM SIGARCH Computer Architecture News*, 44(3):204–216.

Hsieh, K., Khan, S., Vijaykumar, N., Chang, K. K., Boroumand, A., Ghose, S., and Mutlu, O. (2016b). Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *Int. Conf. on Computer Design (ICCD)*.

Huang, Y. et al. (2020). A heterogeneous pim hardware-software co-design for energy-efficient graph processing. In *Int. Parallel and Distributed Processing Symp. (IPDPS)*.

Huangfu, W., Malladi, K. T., Li, S., Gu, P., and Xie, Y. (2020). Nest: Dimm based near-data-processing accelerator for k-mer counting. In *Int. Conf. On Computer Aided Design (ICCAD)*.

Hybrid Memory Cube Consortium (2012). Hybrid memory cube specification rev. 2.0. `http://www.hybridmemorycube.org/`.

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

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

Imani, M., Gupta, S., Kim, Y., and Rosing, T. (2019). Floatpim: In-memory acceleration of deep neural network training with high precision. In *Int. Symp. on Computer Architecture (ISCA)*.

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

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

Jain, S., Ranjan, A., Roy, K., and Raghunathan, A. (2018). Computing in memory with spin-transfer torque magnetic ram. *Trans. on Very Large Scale Integration (VLSI) Systems (TVLSI)*.

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

Kara, K., Alistarh, D., Alonso, G., Mutlu, O., and Zhang, C. (2017). Fpga-accelerated dense linear machine learning: A precision-convergence trade-off. In *Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*.

Kepe, T. R. et al. (2019). Database processing-in-memory: An experimental study. In *Proc. VLDB Endow.*

Khaldi, D. and Chapman, B. (2016). Towards automatic hbm allocation using llvm: a case study with knights landing. In *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*.

Kim, S., Oh, H., Park, C., Cho, S., and Lee, S.-W. (2011). Fast, energy efficient scan inside flash memory ssds. In *Int. Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS VLDB)*.

Kocberber, O., Grot, B., Picorel, J., Falsafi, B., Lim, K., and Ranganathan, P. (2013). Meet the walkers accelerating index traversals for in-memory databases. In *Int. Symp. on Microarchitecture (MICRO)*.

Kvatinsky, S., Belousov, D., Liman, S., Satat, G., Wald, N., Friedman, E. G., Kolodny, A., and Weiser, U. C. (2014). Magic—memristor-aided logic. *Trans. on Circuits and Systems*.

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*.

Kwon, Y.-C., Lee, S. H., Lee, J., Kwon, S.-H., Ryu, J. M., Son, J.-P., Seongil, O., Yu, H.-S., Lee, H., Kim, S. Y., Cho, Y., Kim, J. G., Choi, J., Shin, H.-S., Kim, J., Phuah, B., Kim, H., Song, M. J., Choi, A., Kim, D., Kim, S., Kim, E.-B., Wang, D., Kang, S., Ro, Y., Seo, S., Song, J., Youn, J., Sohn, K., and Kim, N. S. (2021). 25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2tflops programmable computing unit using bank-level parallelism, for machine learning applications. In *Int. Solid-State Circuits Conf. (ISSCC)*.

Lee, V. T., Mazumdar, A., del Mundo, C. C., Alaghi, A., Ceze, L., and Oskin, M. (2018). Application codesign of near-data processing for similarity search. In *Int. Parallel and Distributed Processing Symp. (IPDPS)*.

Lehtonen, E. and Laiho, M. (2009). Stateful implication logic with memristors. In *Int. Symp. on Nanoscale Architectures (ISNA)*.

Li, J., Wang, X., Tumeo, A., Williams, B., Leidel, J. D., and Chen, Y. (2019). Pims: a lightweight processing-in-memory accelerator for stencil computations. In *Proceedings of the International Symposium on Memory Systems*, pages 41–52.

Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., and Jouppi, N. P. (2009). 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*, pages 469–480.

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

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

Liu, J., Zhao, H., Ogleari, M. A., Li, D., and Zhao, J. (2018). Processing-in-memory for energy-efficient neural network training: A heterogeneous approach. In *Int. Symp. on Microarchitecture (MICRO)*.

Loh, G. H., Jayasena, N., Oskin, M., Nutter, M., Roberts, D., Meswani, M., Zhang, D. P., and Ignatowski, M. (2013). A processing in memory taxonomy and a case for studying fixed-function pim. In *Workshop on Near-Data Processing (WoNDP)*.

Long, Y., Lee, E., Kim, D., and Mukhopadhyay, S. (2020). Q-pim: A genetic algorithm based flexible dnn quantization method and application to processing-in-memory platform. In *Design Automation Conf. (DAC)*.

Min, C., Mao, J., Li, H., and Chen, Y. (2019). Neuralhmc: an efficient hmc-based accelerator for deep neural networks. In *Asia and South Pacific Design Automation Conf. (ASPDAC)*.

Mirzadeh, N., Koçberber, Y. O., Falsafi, B., and Grot, B. (2015). Sort vs. hash join revisited for near-memory execution. In *5th Workshop on Architectures and Systems for Big Data (ASBD 2015)*.

Nai, L., Hadidi, R., Sim, J., Kim, H., Kumar, P., and Kim, H. (2017). Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International symposium on high performance computer architecture (HPCA)*, pages 457–468. IEEE.

Nair, R., Antao, S. F., Bertolli, C., Bose, P., Brunheroto, J. R., Chen, T., Cher, C.-Y., Costa, C. H., Doi, J., Evangelinos, C., et al. (2015). Active memory cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3):17–1.

Nayak, S. and Patgiri, R. (2019). A review on role of bloom filter on dna assembly. *IEEE Access*, 7:66939–66954.

Oliveira, G. F. et al. (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*.

Patgiri, R., Nayak, S., and Borgohain, S. K. (2018). Role of bloom filter in big data research: A survey. *International Journal of Advanced Computer Science and Applications*.

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*.

Pugsley, S. H., Deb, A., Balasubramonian, R., and Li, F. (2015). Fixed-function hardware sorting accelerators for near data mapreduce execution. In *Int. Conf. on Computer Design (ICCD)*.

Pugsley, S. H. et al. (2014). NDC: analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads. In *Int. Symp. on Performance Analysis of Systems and Software (ISPASS)*.

Ramanathan, A. K., Kalsi, G. S., Srinivasa, S., Chandran, T. M., Pillai, K. R., Omer, O. J., Narayanan, V., and Subramoney, S. (2020). Look-up table based energy efficient processing in cache support for neural network acceleration. In *Int. Symp. on Microarchitecture (MICRO)*.

Sadredini, E., Rahimi, R., Lenjani, M., Stan, M., and Skadron, K. (2020). Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching. In *Int. Symp. on High Performance Computer Architecture (HPCA)*.

Santos, P. C., de Lima, J. P. C., de Moura, R. F., Ahmed, H., Alves, M. A., Beck, A. C., and Carro, L. (2019a). A technologically agnostic framework for cyber-physical and iot processing-in-memory-based systems simulation. *Microprocessors and Microsystems (MICPRO)*.

Santos, P. C. et al. (2017). Operand size reconfiguration for big data processing in memory. In *Design, Automation & Test in Europe Conf.*

Santos, P. C., Forlin, B. E., and Carro, L. (2021a). Providing plug n'play for processing-in-memory accelerators. In *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 651–656. IEEE.

Santos, P. C., Lima, J. P., Moura, R. F., Alves, M. A., Carro, L., and Beck, A. C. S. (2019b). Solving datapath issues on near-data accelerators. In *Int. Embedded Systems Symp. (IESS)*, IESS '19.

Santos, P. C., Moreira, F. B., Cordeiro, A. S., Santos, S. R., Kepe, T. R., Carro, L., and Alves, M. A. Z. (2021b). Survey on near-data processing: Applications and architectures. *Journal of Integrated Circuits and Systems*, 16(2):1–17.

Santos, S. R. and Alves, M. A. Z. (2022). Impacto da largura do vetor de instruções simd em arquiteturas de processamento próximo à memória. In *XXII Escola Regional de Alto Desempenho da Região Sul (ERAD/RS)*.

Santos, S. R. et al. (2022a). Advancing near-data processing with precise exceptions and efficient data fetching. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

Santos, S. R., Moreira, F. B., Kepe, T. R., and Alves, M. A. Z. (2022b). Advancing database system operators with near-data processing. In *Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*.

Schuiki, F., Schaffner, M., Gürkaynak, F. K., and Benini, L. (2018). A scalable near-memory architecture for training deep neural networks on large in-memory datasets. *IEEE Transactions on Computers*, 68(4):484–497.

Sengupta, S. and Rana, A. (2020). Role of bloom filter in analysis of big data. In *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*, pages 6–9. IEEE.

Seshadri, V., Kim, Y., Fallin, C., Lee, D., Ausavarungnirun, R., Pekhimenko, G., Luo, Y., Mutlu, O., Gibbons, P. B., Kozuch, M. A., et al. (2018). Rowclone: Accelerating data movement and initialization using dram. *arXiv preprint arXiv:1805.03502*.

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

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 (ICCAD)*.

Singh, G., Alser, M., Cali, D. S., Diamantopoulos, D., Gomez-Luna, J., Corporaal, H., and Mutlu, O. (2021). Fpga-based near-memory acceleration of modern data-intensive applications. *IEEE Micro*.

Song, L., Qian, X., Li, H., and Chen, Y. (2017). Pipelayer: A pipelined reram-based accelerator for deep learning. In *Int. Symp. on High Performance Computer Architecture (HPCA)*.

Song, L., Zhuo, Y., Qian, X., Li, H., and Chen, Y. (2018). Graphr: Accelerating graph processing using reram. In *Int. Symp. on High Performance Computer Architecture (HPCA)*.

Sun, Y., Wang, Y., and Yang, H. (2017). Energy-efficient sql query exploiting rram-based process-in-memory structure. In *Non-Volatile Memory Systems and Applications Symp. (NVMSA)*.

Taha, T. M., Hasan, R., Yakopcic, C., and McLean, M. R. (2013). Exploring the design space of specialized multicore neural processors. In *Int. Joint Conference on Neural Networks (IJCNN)*.

Takamaeda-Yamazaki, S., Ueyoshi, K., Ando, K., Uematsu, R., Hirose, K., Ikebe, M., Asai, T., and Motomura, M. (2017). Accelerating deep learning by binarized hardware. In *Asia-Pacific Signal and Information Processing Association Annual Summit and Conf. (APSIPA ASC)*.

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

Tomé, D. G. et al. (2018). Hipe: Hmc instruction predication extension applied on database processing. In *Design, Automation & Test in Europe Conf.*

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)*.

Wang, X., Yu, J., Augustine, C., Iyer, R., and Das, R. (2019). Bit prudent in-cache acceleration of deep convolutional neural networks. In *Int. Symp. on High Performance Computer Architecture (HPCA)*.

Wei, M., Snir, M., Torrellas, J., and Tremaine, R. B. (2005). A near-memory processor for vector, streaming and bit manipulation workloads. Technical report, Dept. of Computer Science, UIUC.

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

Xi, S. L., Augusta, A., Athanassoulis, M., and Idreos, S. (2015). Beyond the wall: Near-data processing for databases. In *Int. Workshop on Data Management on New Hardware (DaMoN)*.

Xie, L., Cai, H., and Yang, J. (2019). Real: Logic and arithmetic operations embedded in rram for general-purpose computing. In *Int. Symp. on Nanoscale Architectures (NANOARCH)*.

Xin, X., Zhang, Y., and Yang, J. (2020). Elp2im: Efficient and low power bitwise operation processing in dram. In *Int. Symp. on High Performance Computer Architecture (HPCA)*.

Yin, S., Jiang, Z., Kim, M., Gupta, T., Seok, M., and Seo, J.-S. (2019). Vesti: Energy-efficient in-memory computing accelerator for deep neural networks. *Trans. on Very Large Scale Integration (VLSI) Systems*.

Zhang, D., Jayasena, N., Lyashevsky, A., Greathouse, J. L., Xu, L., and Ignatowski, M. (2014). Top-pim: Throughput-oriented programmable processing in memory. In *Int. Symp. on High-performance Parallel and Distributed Computing (HPDC)*.

Zhang, D. P., Jayasena, N., Lyashevsky, A., Greathouse, J., Meswani, M., Nutter, M., and Ignatowski, M. (2013). A new perspective on processing-in-memory architecture design. In *SIGPLAN Workshop on Memory Systems Performance and Correctness*.

Zhu, Q., Akin, B., Sumbul, H. E., Sadi, F., Hoe, J. C., Pileggi, L., and Franchetti, F. (2013a). A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. In *Int. 3D Systems Integration Conf. (3DIC)*.

Zhu, Q., Graf, T., Sumbul, H. E., Pileggi, L., and Franchetti, F. (2013b). Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware. In *High Performance Extreme Computing Conf. (HPEC)*.

APPENDIX A – TABLE OF INTRINSICS-VIMA INSTRUCTIONS

Table A.1 describes all instructions implemented in the Intrinsics-VIMA library.

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

Start of Table		
Function	Data type	Description
VIMA Arithmetic Instructions - Integer		
`_vim2K_iadds(*a, *b, *c)`	`__v32s`	Performs signed addition between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_iaddu(*a, *b, *c)`	`__v32u`	Performs unsigned addition between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_isubs(*a, *b, *c)`	`__v32s`	Performs signed subtraction between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_isubu(*a, *b, *c)`	`__v32u`	Performs unsigned subtraction between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_iabss(*a, *b)`	`__v32s`	Takes the absolute value of each 32-bit element in a source vector A[0:2047] and stores it into the destination vector B[0:2047].
`_vim2K_imaxs(*a, *b, *c)`	`__v32s`	Find the maximal value between each 32-bit element of source vectors A[0:2047] and B[0:2047] and stores it into the destination vector C[0:2047].
`_vim2K_imins(*a, *b, *c)`	`__v32s`	Find the minimal value between each 32-bit element of source vectors A[0:2047] and B[0:2047] and stores it into the destination vector C[0:2047].
`_vim2K_icpys(*a, *b)`	`__v32s`	Performs signed copy of 32-bit element of source vectors A[0:2047] and stores it into the destination vector B[0:2047].
`_vim2K_icpyu(*a, *b)`	`__v32u`	Performs unsigned copy of 32-bit element of source vectors A[0:2047] and stores it into the destination vector B[0:2047].
VIMA Arithmetic Instructions - Floating-point Single Precision		
`_vim2K_fadds(*a, *b, *c)`	`__v32f`	Performs signed addition between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_fsubs(*a, *b, *c)`	`__v32f`	Performs signed subtraction between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_fabss(*a, *b)`	`__v32f`	Takes the absolute value of each 32-bit element in a source vector A[0:2047] and stores it into the destination vector B[0:2047].
`_vim2K_fmaxs(*a, *b, *c)`	`__v32f`	Find the maximal value between each 32-bit element of source vectors A[0:2047] and B[0:2047] and stores it into the destination vector C[0:2047].
`_vim2K_fmins(*a, *b, *c)`	`__v32f`	Find the minimal value between each 32-bit element of source vectors A[0:2047] and B[0:2047] and stores it into the destination vector C[0:2047].
`_vim2K_fcpys(*a, *b)`	`__v32f`	Performs signed copy of 32-bit element of source vectors A[0:2047] and stores it into the destination vector B[0:2047].

Continuation of Table A.1		
Function	Data type	Description
VIMA Arithmetic Instructions - Floating-point Double Precision		
`_vim1K_dadds(*a, *b, *c)`	`__v64d`	Performs signed addition between 64-bit elements source vectors A[0:1023] and B[0:1023] and stores the result into the destination vector C[0:1023].
`_vim1K_dsubs(*a, *b, *c)`	`__v64d`	Performs signed subtraction between 64-bit elements source vectors A[0:1023] and B[0:1023] and stores the result into the destination vector C[0:1023].
`_vim1K_dabss(*a, *b)`	`__v64d`	Takes the absolute value of each 64-bit element in a source vector A[0:1023] and stores it into the destination vector B[0:1023].
`_vim1K_dmaxs(*a, *b, *c)`	`__v64d`	Find the maximal value between each 64-bit element of source vectors A[0:1023] and B[0:1023] and stores it into the destination vector C[0:1023].
`_vim1K_dmins(*a, *b, *c)`	`__v64d`	Find the minimal value between each 64-bit element of source vectors A[0:1023] and B[0:1023] and stores it into the destination vector C[0:1023].
`_vim1K_dcpys(*a, *b)`	`__v64d`	Performs signed copy of 64-bit element of source vectors A[0:1023] and stores it into the destination vector B[0:1023].
VIMA Logic Instructions - Integer		
`_vima2K_iandu(*a, *b, *c)`	`__vm32u`	Performs AND operation between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_iandu(*a, *b, *c)`	`__v32u`	Performs AND operation between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_iorun(*a, *b, *c)`	`__v32u`	Performs OR operation between 32-bit elements of source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_ixoru(*a, *b, *c)`	`__v32u`	Performs XOR operation between 32-bit elements source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_inots(*a, *b)`	`__v32s`	Performs NOT operation in 32-bit elements source vector A[0:2047] and stores the result into the destination vector B[0:2047].
VIMA Comparison Instructions - Integer		
`_vim2K_islts(*a, *b, *c)`	`__v32s`	Compare each signed 32-bit elements from source vectors A[0:2047] and B[0:2047] and if the element of A[0:2047] is minor, then destination source C[0:2047] stores 1 in the same position, otherwise, stores 0.
`_vim2K_isltu(*a, *b, *c)`	`__v32u`	Compare each unsigned 32-bit elements from source vectors A[0:2047] and B[0:2047] and if the element of A[0:2047] is minor, then destination source C[0:2047] stores 1 in the same position, otherwise, stores 0.
`_vim2K_icmqs(*a, *b, *c)`	`__v32s`	Compare each signed 32-bit elements from source vectors A[0:2047] and B[0:2047] and if they are equal, then destination source C[0:2047] stores 1 in the same position, otherwise, stores 0.
`_vim2K_icmqu(*a, *b, *c)`	`__v32u`	Compares each unsigned 32-bit elements from source vectors A[0:2047] and B[0:2047] and if they are equal, then destination source C[0:2047] stores 1 in the same position, otherwise, stores 0.
VIMA Comparison Instructions - Floating-point Single Precision		

Continuation of Table A.1		
Function	Data type	Description
`_vim2K_fslts(*a, *b, *c)`	`__v32f`	Compare each signed 32-bit elements from source vectors A[0:2047] and B[0:2047] and if the element of A[0:2047] is minor, then destination source C[0:2047] stores 1 in the same position, otherwise, stores 0.
`_vim2K_fcmqs(*a, *b, *c)`	`__v32f`	Compare each signed 32-bit elements from source vectors A[0:2047] and B[0:2047] and if they are equal, then destination source C[0:2047] stores 1 in the same position, otherwise, stores 0.
VIMA Comparison Instructions - Floating-point Double Precision		
`_vim1K_dslts(*a, *b, *c)`	`__v64d`	Compare each signed 64-bit elements from source vectors A[0:1023] and B[0:1023] and if the element of A[0:1023] is minor, then destination source C[0:1023] stores 1 in the same position, otherwise, stores 0.
`_vim1K_dcmqs(*a, *b, *c)`	`__v64d`	Compare each signed 64-bit elements from source vectors A[0:1023] and B[0:1023] and if they are equal, then destination source C[0:1023] stores 1 in the same position, otherwise, stores 0.
VIMA Shift Instructions - Integer		
`_vim2K_isllu(*a, *b, *c)`	`__v32u`	Left shift each 32-bit element in source vector A[0:2047] the amount specified in source vector B[0:2047] and stores the result into the destination vector C[0:2047]. This operation does not shift signal.
`_vim2K_isrlu(*a, *b, *c)`	`__v32u`	Right shift each 32-bit element in source vector A[0:2047] the amount specified in source vector B[0:2047] and stores the result into the destination vector C[0:2047]. This operation does not shift signal.
`_vim2K_isras(*a, *b, *c)`	`__v32s`	Right shift each 32-bit element in source vector A[0:2047] the amount specified in source vector B[0:2047] and stores the result into the destination vector C[0:2047]. This operation shifts signal.
VIMA Multiplication/Division Instructions - Integer		
`_vim2K_idivs(*a, *b, *c)`	`__v32s`	Performs a signed division between 32-bit elements from source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_idivu(*a, *b, *c)`	`__v32u`	Performs an unsigned division between 32-bit elements from source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_imuls(*a, *b, *c)`	`__v32s`	Performs signed multiplication between 32-bit elements from source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_imulu(*a, *b, *c)`	`__v32u`	Performs unsigned multiplication between 32-bit elements from source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim1K_imuls(*a, *b, *c)`	`__v64s`	Performs signed multiplication between 64-bit elements from source vectors A[0:1023] and B[0:1023] and stores the result into the destination vector C[0:1023].
`_vim1K_imulu(*a, *b, *c)`	`__v64u`	Performs unsigned multiplication between 64-bit elements from source vectors A[0:1023] and B[0:1023] and stores the result into the destination vector C[0:1023].
`_vim2K_icums(*a, *b)`	`__v32s`	Performs signed accumulated sum between 32-bit elements from source vector A[0:2047] and stores the result into the destination variable b.
`_vim2K_icumu(*a, *b)`	`__v32u`	Performs unsigned accumulated sum between 32-bit elements from source vector A[0:2047] and stores the result into the destination variable b.

Continuation of Table A.1		
Function	Data type	Description
VIMA Multiplication/Division Instructions - Floating-point Single Precision		
`_vim2K_fdivs(*a, *b, *c)`	`__v32f`	Performs a signed division between 32-bit elements from source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_fmuls(*a, *b, *c)`	`__v32f`	Performs signed multiplication between 32-bit elements from source vectors A[0:2047] and B[0:2047] and stores the result into the destination vector C[0:2047].
`_vim2K_fcums(*a, *b)`	`__v32f`	Performs signed accumulated sum between 32-bit elements from source vector A[0:2047] and stores the result into the destination variable b.
VIMA Multiplication/Division Instructions - Floating-point Double Precision		
`_vim1K_ddivs(*a, *b, *c)`	`__v64d`	Performs a signed division between 64-bit elements from source vectors A[0:1023] and B[0:1023] and stores the result into the destination vector C[0:1023].
`_vim1K_dmuls(*a, *b, *c)`	`__v64d`	Performs signed multiplication between 64-bit elements from source vectors A[0:1023] and B[0:1023] and stores the result into the destination vector C[0:1023].
`_vim1K_dcums(*a, *b)`	`__v64d`	Performs signed accumulated sum between 64-bit elements from source vector A[0:1023] and stores the result into the destination variable b.
VIMA Immediate Instructions - Integer		
`_vim2K_imovs(*a, *b)`	`__v32s`	Replicate a signed 32-bit immediate b into the vector A[0:2047].
`_vim2K_imovu(*a, *b)`	`__v32u`	Replicate a unsigned 32-bit immediate b into the vector A[0:2047].
VIMA Immediate Instructions - Floating-point Single Precision		
`_vim2K_fmovs(*a, *b)`	`__v32f`	Replicate a signed 32-bit immediate b into the vector A[0:2047].
VIMA Immediate Instructions - Floating-point Double Precision		
`_vim1K_dmovs(*a, *b)`	`__v64d`	Replicate a signed 64-bit immediate b into the vector A[0:1023].
VIMA Mask Instructions - Integer		
`_vim2K_ilmks(*a, *b, *c)`	`__v32s`	Loads signed 32-bit elements into vector A[0:2047] and stores values into the destination vector C[0:2047] according to the mask on vector B[0:2047].
`_vim2K_ilmku(*a, *b, *c)`	`__v32u`	Loads unsigned 32-bit elements into vector A[0:2047] and stores values into the destination vector C[0:2047] according to the mask on vector B[0:2047].
`_vim2K_irmks(*a, *b, *c)`	`__v32s`	Loads signed 32-bit elements into vector A[0:2047] and resets values to zero in the destination vector C[0:2047] according to the mask on vector B[0:2047]; remaining elements are copied.
`_vim2K_irmku(*a, *b, *c)`	`__v32u`	Loads unsigned 32-bit elements into vector A[0:2047] and resets values to zero in the destination vector C[0:2047] according to the mask on vector B[0:2047]; remaining elements are copied.
VIMA Multiplication/Division Instructions - Floating-point Single Precision		
`_vim2K_flmks(*a, *b, *c)`	`__v32f`	Loads signed 32-bit elements into vector A[0:2047] and stores values into the destination vector C[0:2047] according to the mask on vector B[0:2047].
`_vim2K_frmks(*a, *b, *c)`	`__v32f`	Loads signed 32-bit elements into vector A[0:2047] and resets values to zero in the destination vector C[0:2047] according to the mask on vector B[0:2047]; remaining elements are copied.
VIMA Multiplication/Division Instructions - Floating-point Double Precision		

Continuation of Table A.1

Function	Data type	Description
`_vim1K_dlmks(*a, *b, *c)`	`__v64d`	Loads signed 64-bit elements into vector A[0:1023] and stores values into the destination vector C[0:1023] according to the mask on vector B[0:1023].
`_vim1K_drmks(*a, *b, *c)`	`__v64d`	Loads signed 64-bit elements into vector A[0:1023] and resets values to zero in the destination vector C[0:1023] according to the mask on vector B[0:1023]; remaining elements are copied.

End of Table

APPENDIX B – DETAILED EXPERIMENT RESULTS

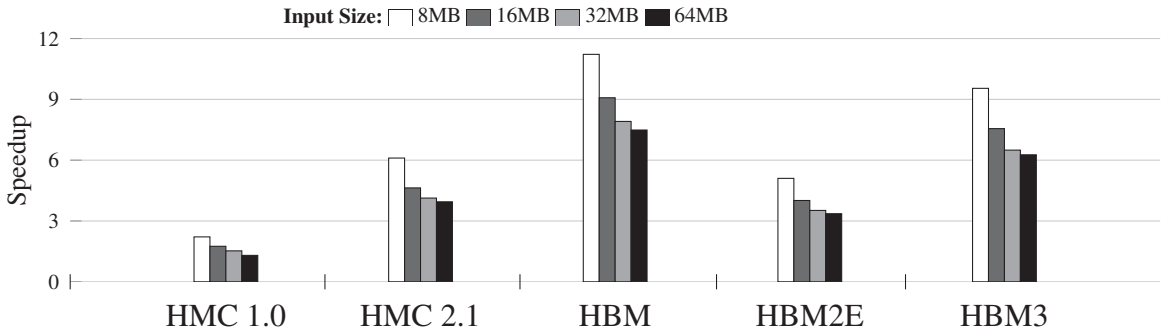


Figure B.1: Execution time results executing *memory copy application* with VIMA with perfect interconnection and request size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

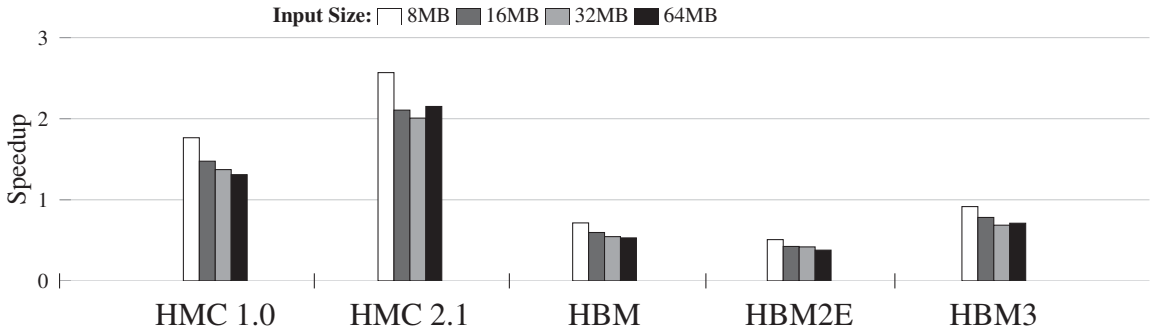


Figure B.2: Execution time results executing *vector sum application* with VIMA with perfect interconnection and request size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

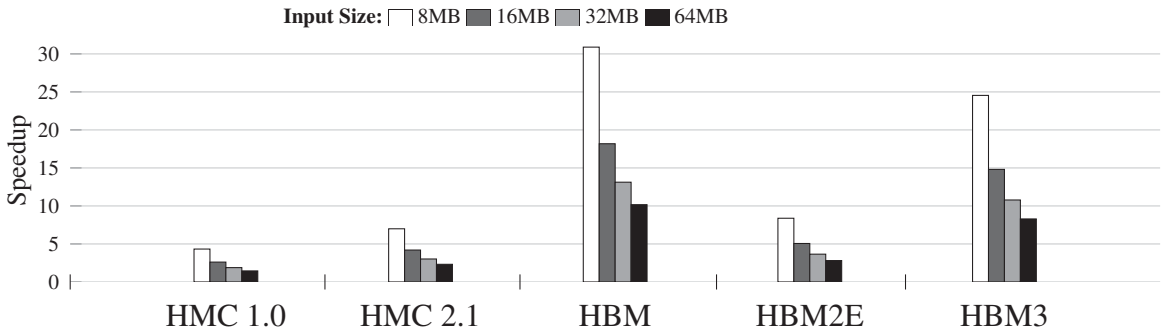


Figure B.3: Execution time results executing *selection database query* with VIMA with perfect interconnection and request size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

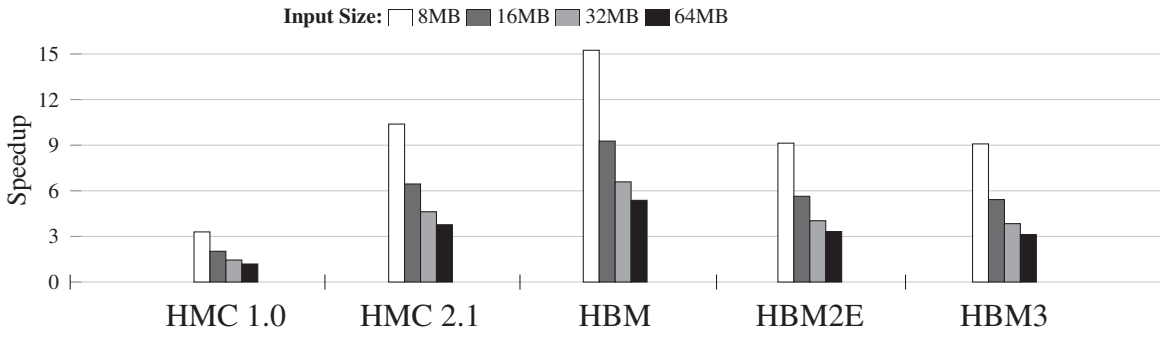


Figure B.4: Execution time results executing *projection database query* with VIMA with perfect interconnection and request size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

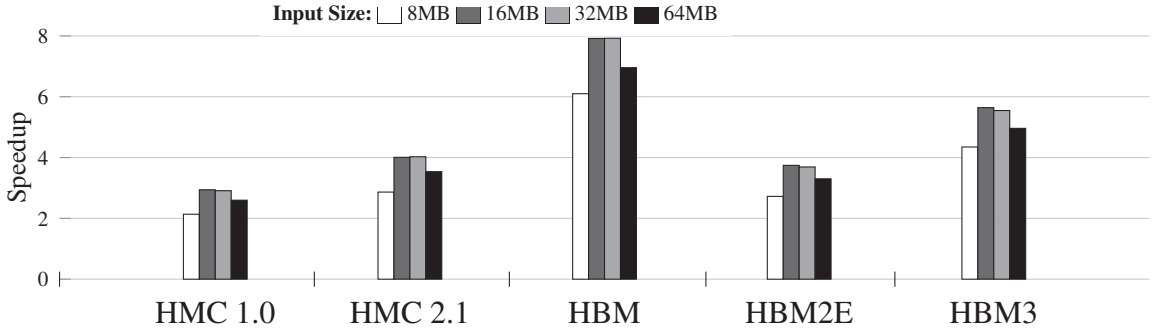


Figure B.5: Execution time results executing *stencil application* with VIMA with perfect interconnection and request size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

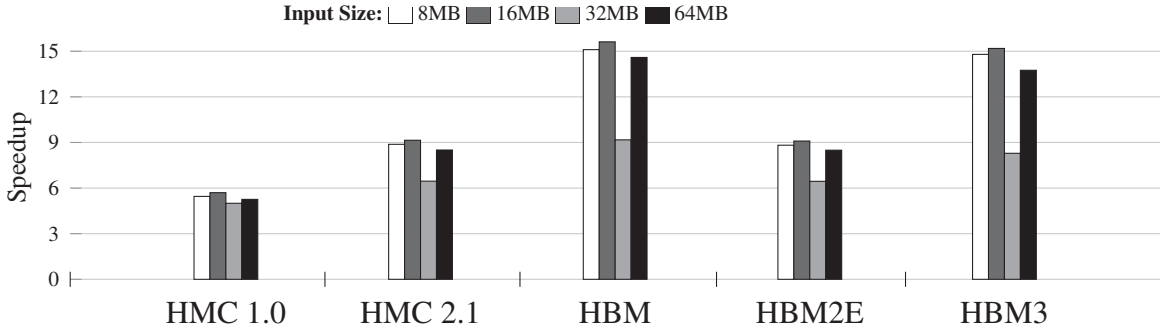


Figure B.6: Execution time results executing *bloom filter application* with VIMA with perfect interconnection and request size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

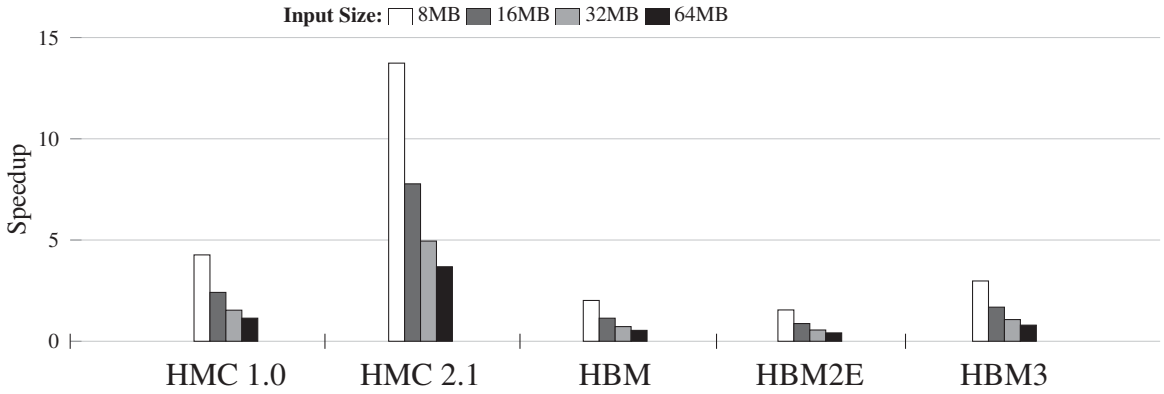


Figure B.7: Execution time results executing *memory set application* with VIMA in the maximum specified request and interconnection size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

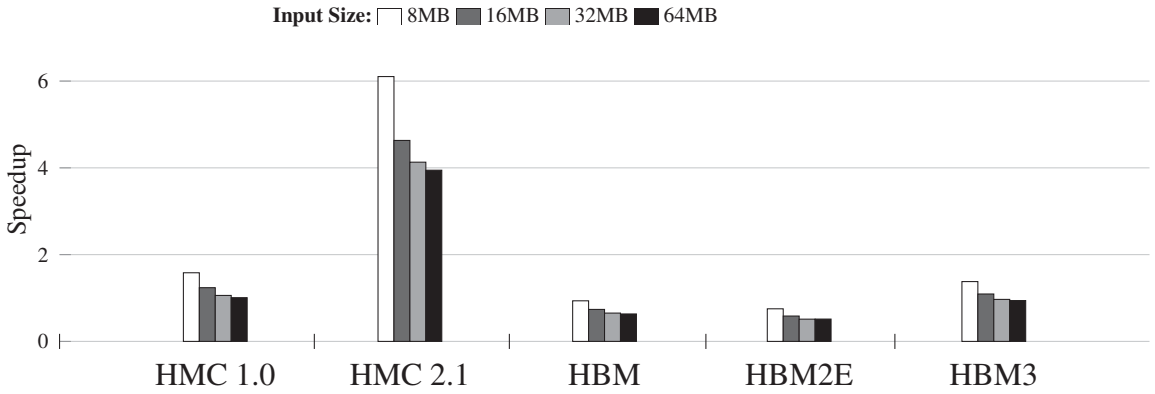


Figure B.8: Execution time results executing *memory copy application* with VIMA in the maximum specified request and interconnection size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

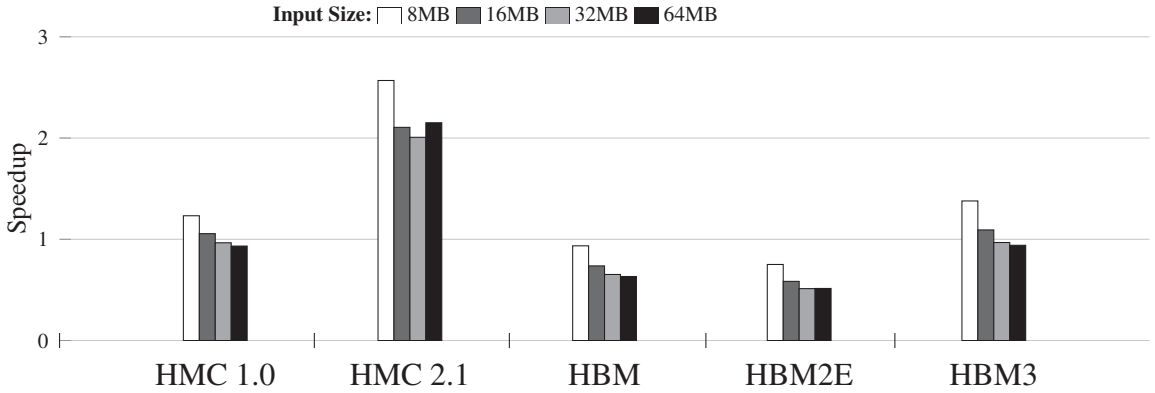


Figure B.9: Execution time results executing *vector sum application* with VIMA in the maximum specified request and interconnection size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

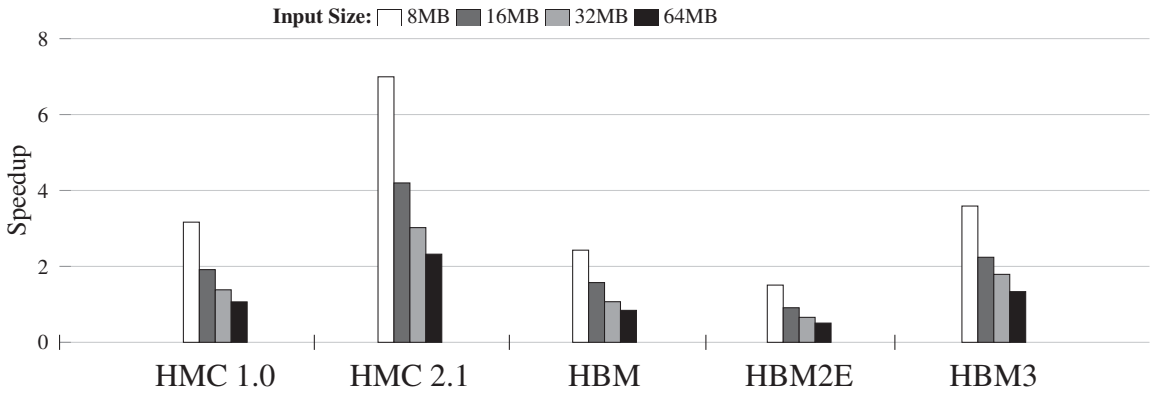


Figure B.10: Execution time results executing *selection database query* with VIMA in the maximum specified request and interconnection size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

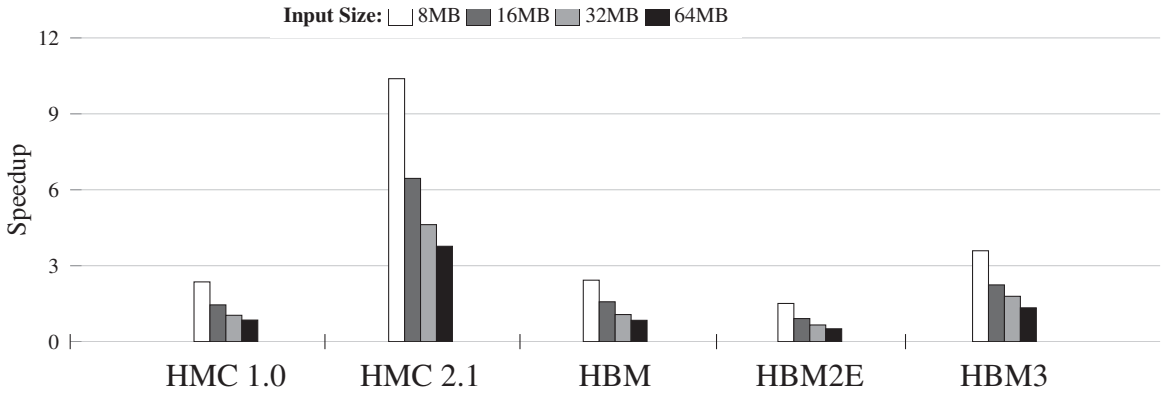


Figure B.11: Execution time results executing *projection database query* with VIMA in the maximum specified request and interconnection size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

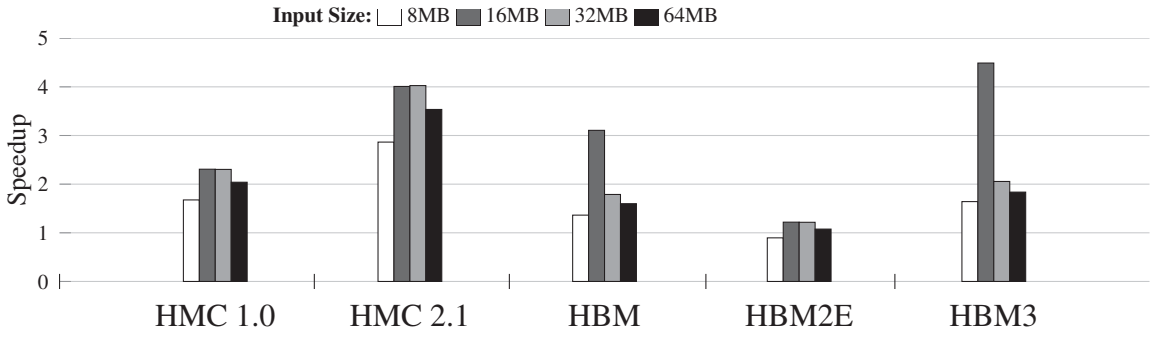


Figure B.12: Execution time results executing *stencil application* with VIMA in the maximum specified request and interconnection size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

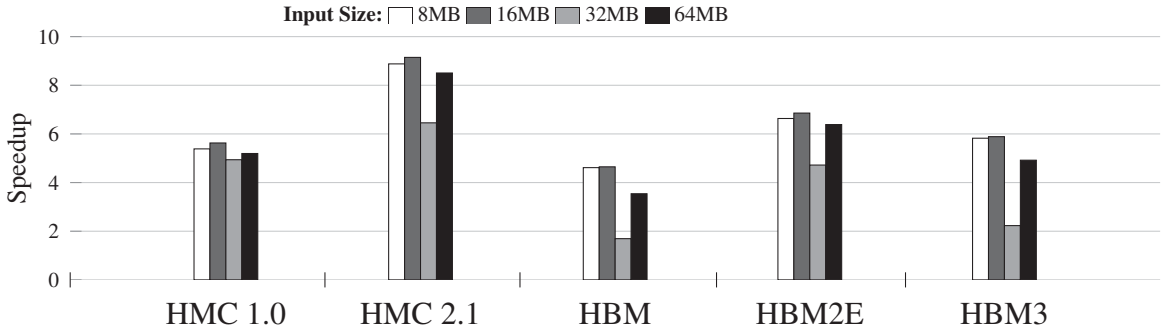


Figure B.13: Execution time results executing *bloom filter application* with VIMA in the maximum specified request and interconnection size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

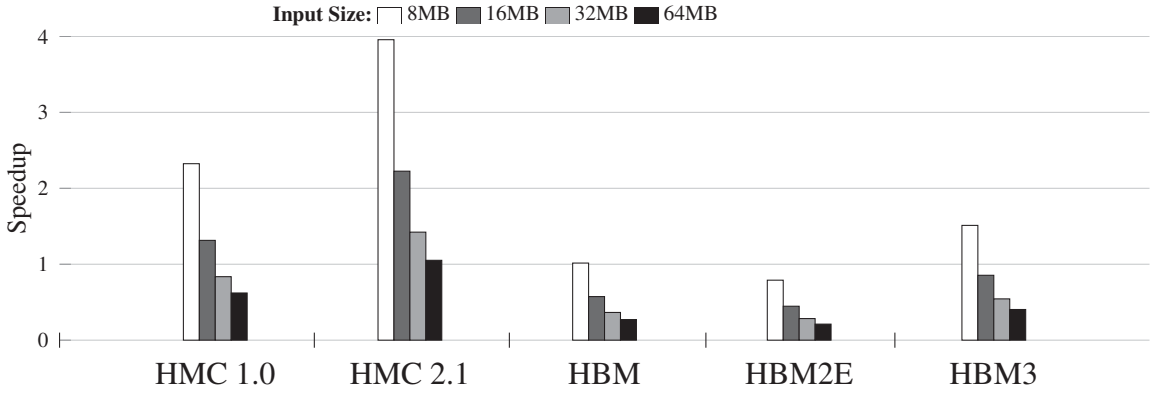


Figure B.14: Execution time results executing *memory set application* with VIMA in a 64 B interconnection and request size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

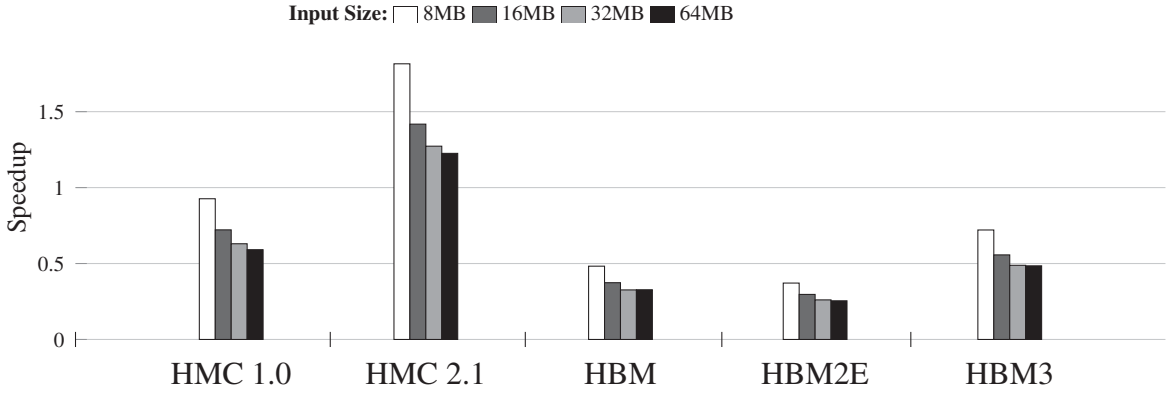


Figure B.15: Execution time results executing *memory copy application* with VIMA in a 64 B interconnection and request size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

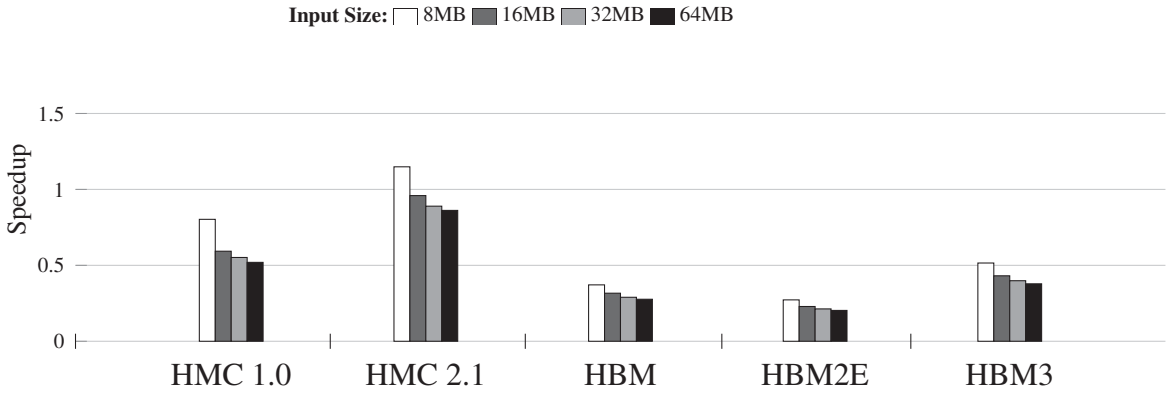


Figure B.16: Execution time results executing *vector sum application* with VIMA in a 64 B interconnection and request size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

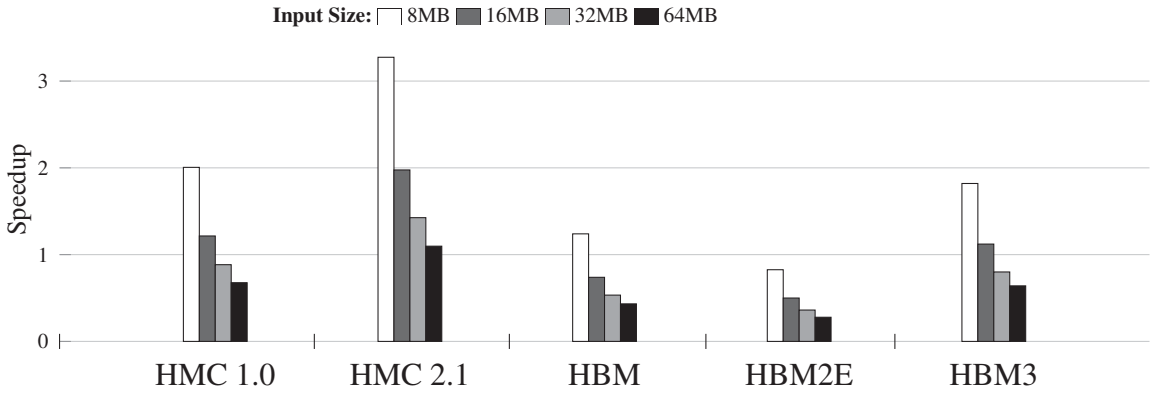


Figure B.17: Execution time results executing *selection database query* with VIMA in a 64 B interconnection and request size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

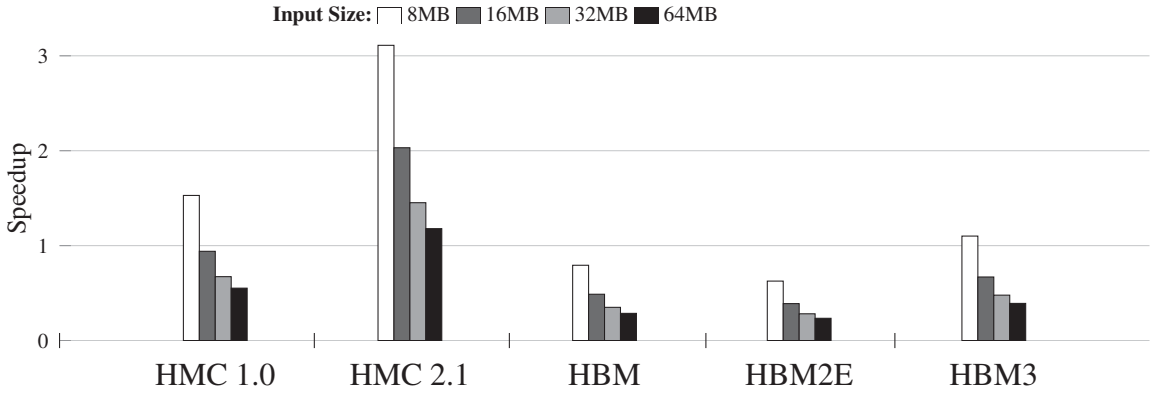


Figure B.18: Execution time results executing *projection database query* with VIMA in a 64 B interconnection and request size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

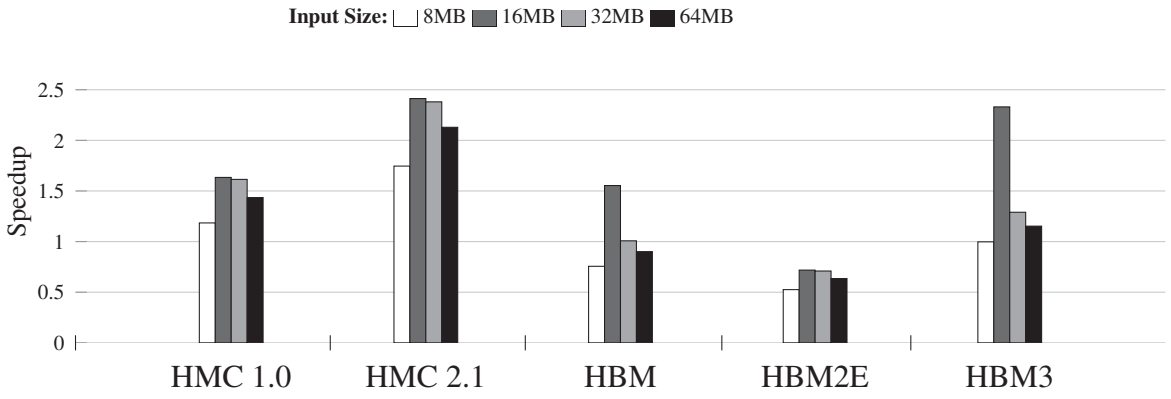


Figure B.19: Execution time results executing *stencil application* with VIMA in a 64 B interconnection and request size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

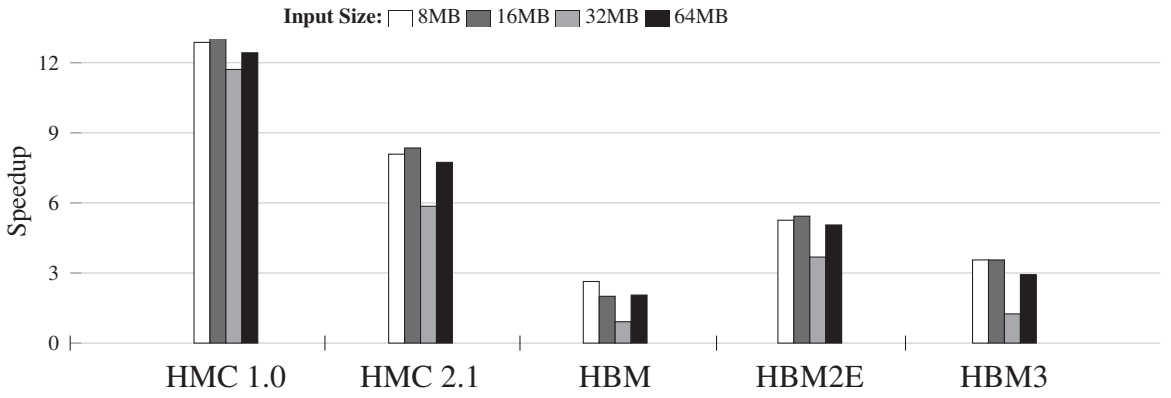


Figure B.20: Execution time results executing *bloom filter application* with VIMA in a 64 B interconnection and request size scenario, normalized to 16-thread x86 baseline. Values higher than 1 indicate improvement in performance over the baseline.

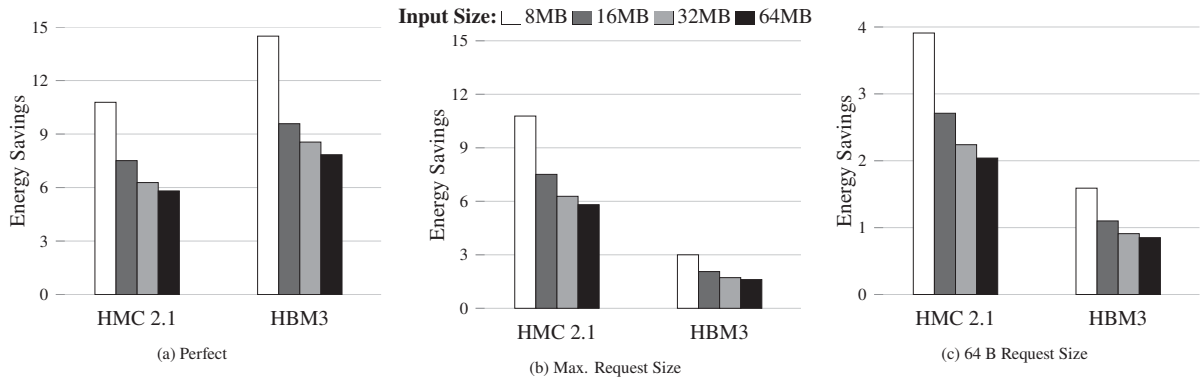


Figure B.21: Energy savings of VIMA baseline running over *memory set application* for (a) Perfect access, (b) Access considering the maximum request size supported by each 3D-stacked memory and (c) Access in 64 B requests. Values higher than 1 indicate improvement in performance over the baseline.

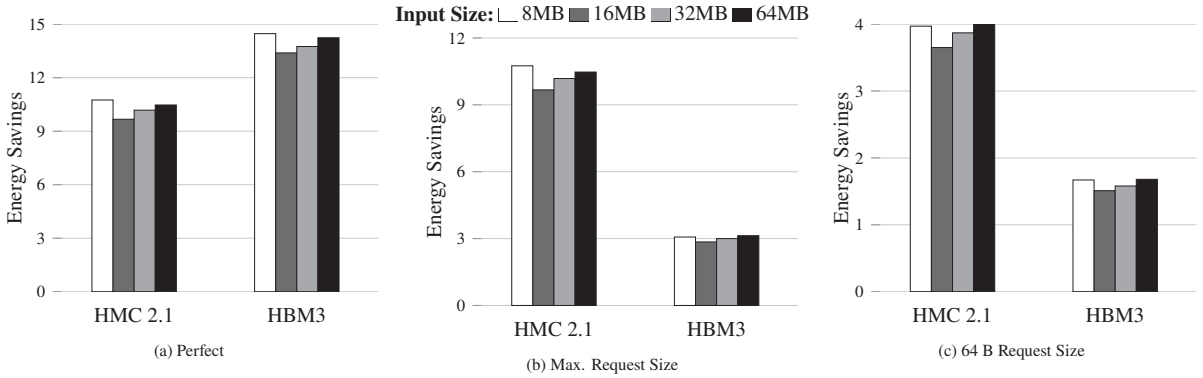


Figure B.22: Energy savings of VIMA over baseline running *memory copy application* for (a) Perfect access, (b) Access considering the maximum request size supported by each 3D-stacked memory and (c) Access in 64 B requests. Values higher than 1 indicate improvement in performance over the baseline.

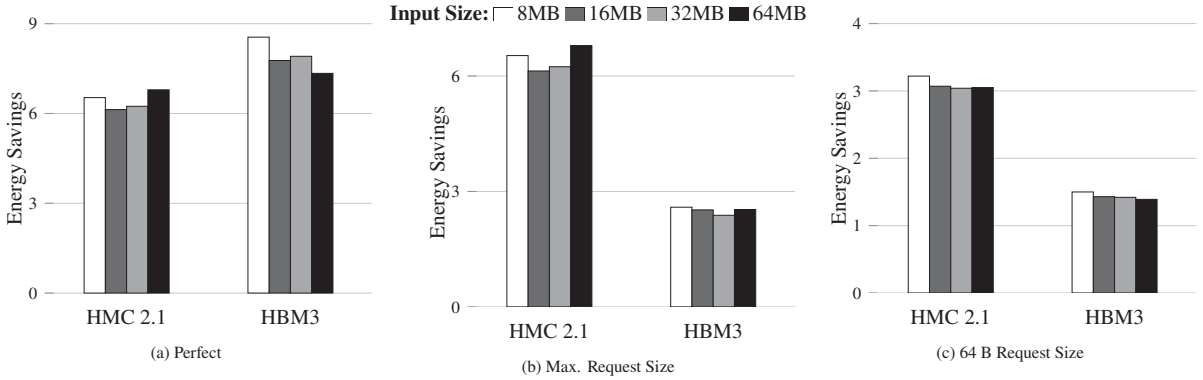


Figure B.23: Energy savings of VIMA running over baseline *vector sum application* for (a) Perfect access, (b) Access considering the maximum request size supported by each 3D-stacked memory and (c) Access in 64 B requests. Values higher than 1 indicate improvement in performance over the baseline.

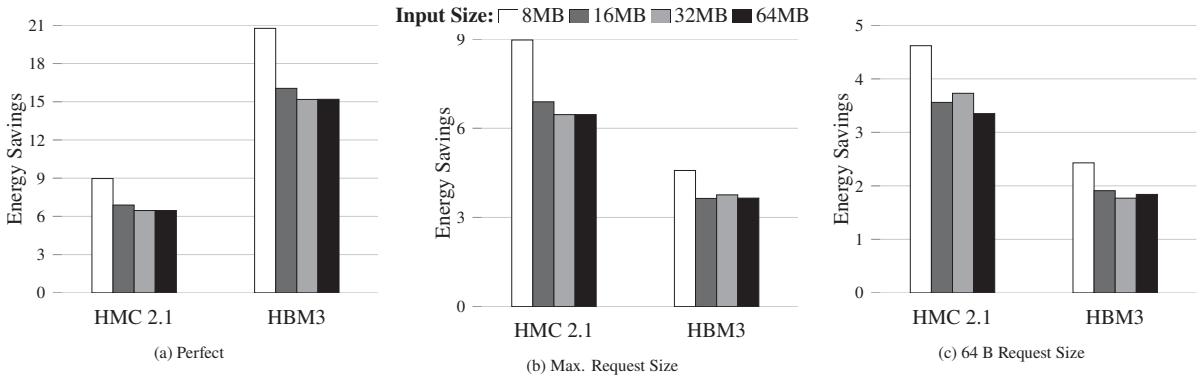


Figure B.24: Energy savings of VIMA over baseline running *selection database query* for (a) Perfect access, (b) Access considering the maximum request size supported by each 3D-stacked memory and (c) Access in 64 B requests. Values higher than 1 indicate improvement in performance over the baseline.

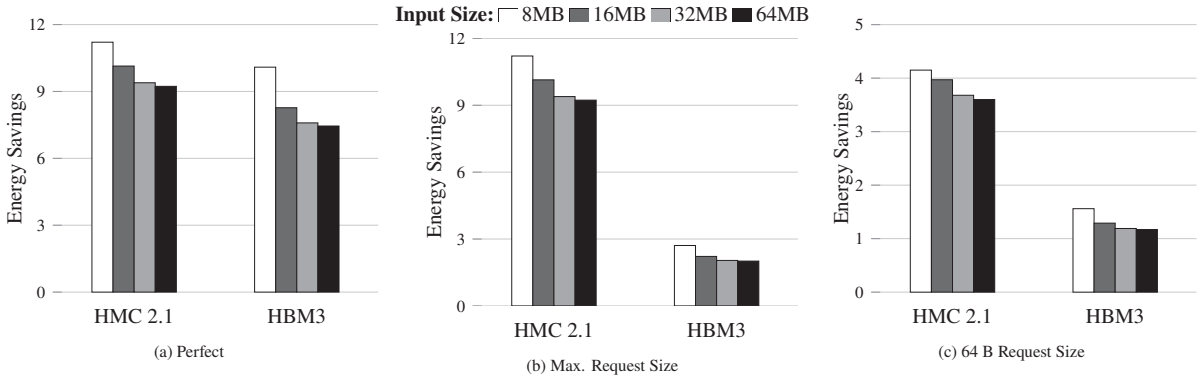


Figure B.25: Energy savings of VIMA over baseline running *projection database query* for (a) Perfect access, (b) Access considering the maximum request size supported by each 3D-stacked memory and (c) Access in 64 B requests. Values higher than 1 indicate improvement in performance over the baseline.

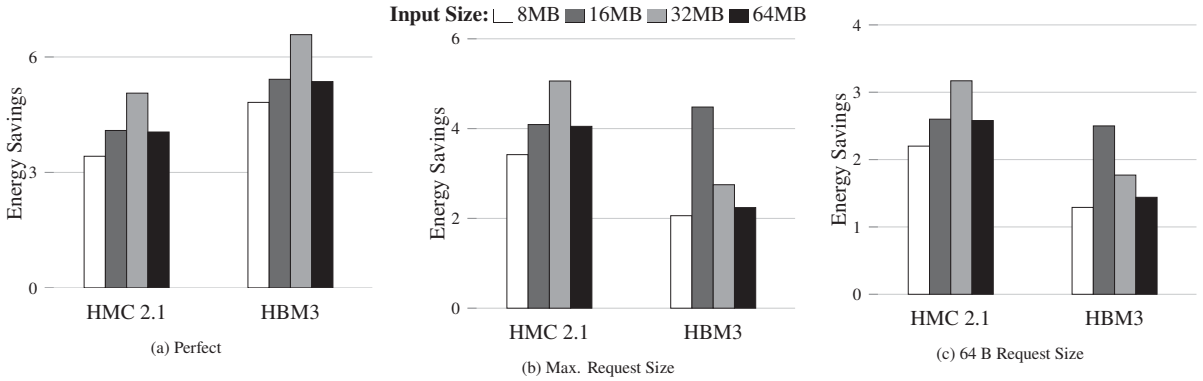


Figure B.26: Energy savings of VIMA over baseline running *stencil application* for (a) Perfect access, (b) Access considering the maximum request size supported by each 3D-stacked memory and (c) Access in 64 B requests. Values higher than 1 indicate improvement in performance over the baseline.

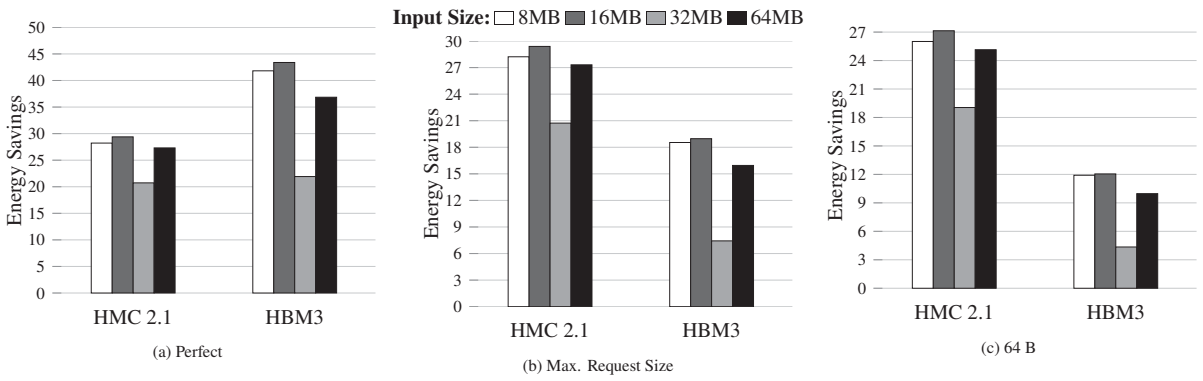


Figure B.27: Energy savings of VIMA over baseline running *bloom filter application* for (a) Perfect access, (b) Access considering the maximum request size supported by each 3D-stacked memory and (c) Access in 64 B requests. Values higher than 1 indicate improvement in performance over the baseline.

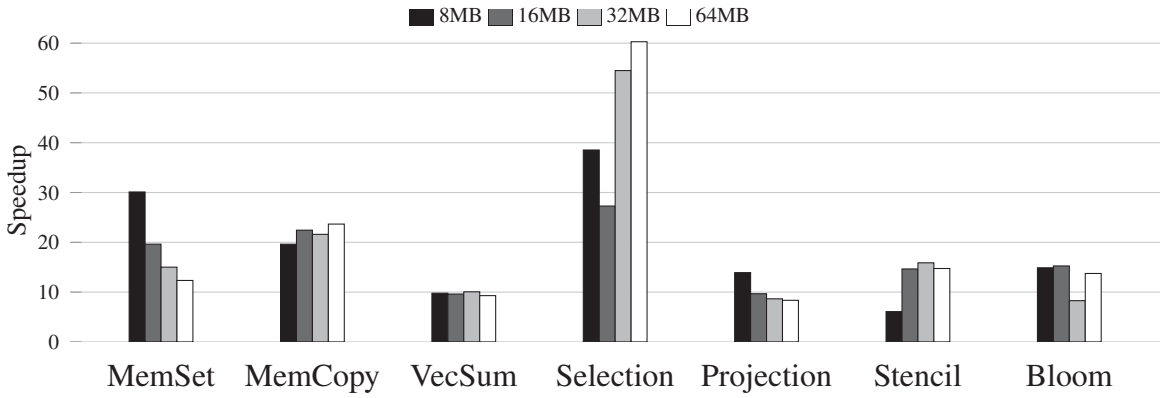


Figure B.28: Execution time results of VIMA executing all workloads with perfect access to 3D-stacked memory row buffers normalized to 16-thread x86 baseline running with a HBM3 memory. Values higher than 1 indicate improvement in performance over the baseline.

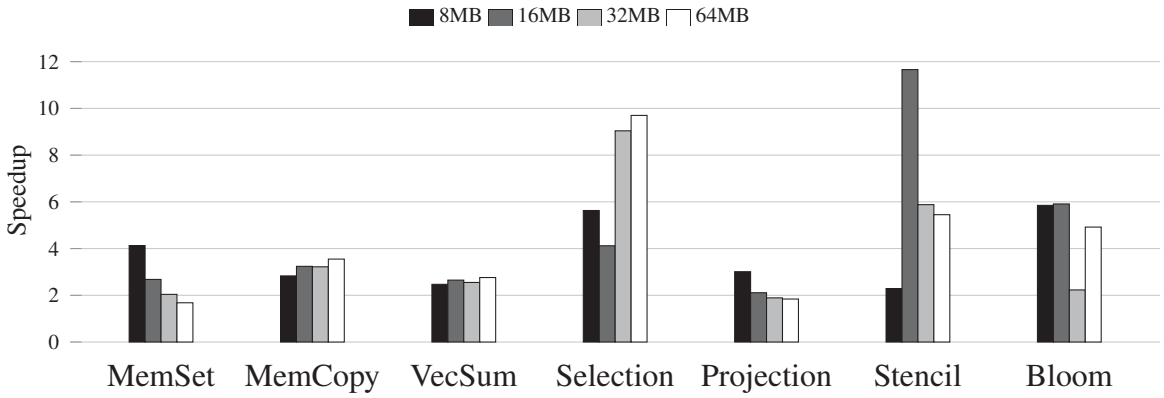


Figure B.29: Execution time results of VIMA executing all workloads with maximum request size normalized to 16-thread x86 baseline running with a HBM3 memory. Values higher than 1 indicate improvement in performance over the baseline.

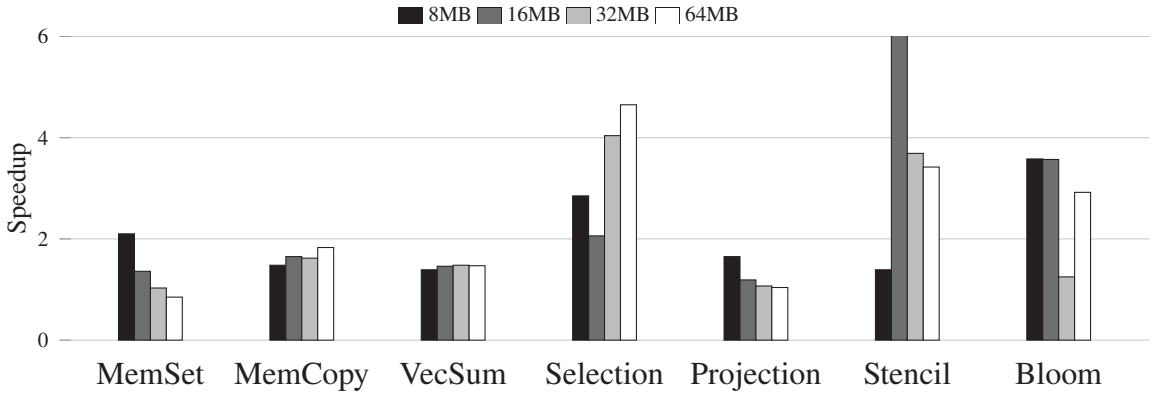


Figure B.30: Execution time results of VIMA executing all workloads with 64B request size normalized to 16-thread x86 baseline running with a HBM3 memory. Values higher than 1 indicate improvement in performance over the baseline.

APPENDIX C – APPLICATION CODE WITH INTRINSICS-VIMA

Code C.1 shows the code using the library Intrinsics-VIMA, which we used to generate the simulation trace used for our experiments.

Code C.1: Intrinsics-VIMA routine call for memory set.

```
uint32_t vima_size = 2048;

// Allocate the vector
__v32f *vector = (__v32s*)malloc(sizeof(__v32s) * vima_size * x);

// Initialize the memory locations
<...>

// Perform the memory setting: vector[i] = 1
for (int i = 0; i < vima_size * x; i += vima_size) {
   _vim2K_imovs(1, &vector[i]);
}
```

Code C.2 shows the implementation of the *memory copy* application and Code C.3 shows the code for the *vector sum* application.

Code C.2: Intrinsics-VIMA routine call for memory copy.

```
uint32_t vima_size = 2048;

// Allocate the vectors A, B (sources) and C (destination)
__v32s *A = (__v32s *) malloc (32, sizeof(__v32s) * v_size * x);
__v32s *B = (__v32s *) malloc (32, sizeof(__v32s) * v_size * x);

// Initialize the memory locations
<...>

// Perform the memory copying: B[i] = A[i]
for (int i = 0; i < vima_size * x; i += vima_size) {
   _vim2K_icpys (&A[i], &B[i]);
}
```

Code C.3: Intrinsics-VIMA routine call for vector sum.

```
uint32_t vima_size = 2048;

// Allocate the vectors A, B (sources) and C (destination)
__v32f *A = (__v32f *) malloc (32, sizeof(__v32f) * v_size * x);
__v32f *B = (__v32f *) malloc (32, sizeof(__v32f) * v_size * x);
__v32f *C = (__v32f *) malloc (32, sizeof(__v32f) * v_size * x);

// Initialize the memory locations
<...>

// 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]);
}
```

Codes C.4 and C.5 show the code used to generate the simulation traces used in our experiments.

Code C.4: Intrinsics-VIMA routine call for the selection database query operator.

```
uint32_t vima_size = 2048;

// Allocate the vectors A, B (sources) and C (destination)
__v32f *filter = (__v32f *) malloc (32, sizeof(__v32f) * v_size);
__v32f *A = (__v32f *) malloc (32, sizeof(__v32f) * v_size * x);
__v32f *bitmap = (__v32f *) malloc (32, sizeof(__v32f) * v_size * x);

// Initialize the memory locations and filter value
```

```
<...>

// Perform the selection according to filter value:
for (int i = 0; i < vima_size * x; i += vima_size) {
   _vim2K_isltu (filter, &A[i], &bitmap[i]);
}
```

Code C.5: Intrinsics-VIMA routine call for the projection database query operator.

```
uint32_t vima_size = 2048;

// Allocate the vectors A, B (sources) and C (destination)
__v32f *vector = (__v32f *) malloc (32, sizeof(__v32f) * v_size);
__v32f *bitmap = (__v32f *) malloc (32, sizeof(__v32f) * v_size * x);
__v32f *result = (__v32f *) malloc (32, sizeof(__v32f) * v_size * x);

// Initialize the memory locations
<...>

// Perform the loading according to input bitmap:
for (int i = 0; i < vima_size * x; i += vima_size) {
   _vim2K_ilmku (&vector[i], &bitmap[i], &result[i]);
}
```

Code C.6: Intrinsics-VIMA routine for the stencil application.

```
uint32_t vima_size = 2048;

// Allocate the vectors A, B (sources) and C (destination)
__v32f *vector_a = (__v32f *) malloc (32, sizeof(__v32f) * v_size);
__v32f *vector_b = (__v32f *) malloc (32, sizeof(__v32f) * v_size * x);
__v32f *mul = (__v32f *) malloc (32, sizeof(__v32f) * v_size * x);

// Initialize the memory locations
<...>

for (int i = elem; i+elem+VECTOR_SIZE < v_size; i += VECTOR_SIZE) {
   _vim2K_fadds(&vector_b[i], &vector_a[i-elem], &vector_b[i]);
   _vim2K_fadds(&vector_b[i], &vector_a[i], &vector_b[i]);
   _vim2K_fadds(&vector_b[i], &vector_a[i-1], &vector_b[i]);
   _vim2K_fadds(&vector_b[i], &vector_a[i+1], &vector_b[i]);
   _vim2K_fadds(&vector_b[i], &vector_a[i+elem], &vector_b[i]);
   _vim2K_fmuls(&vector_b[i], &mul[i], &vector_b[i]);
   remainder = i;
}
```

Code C.7: Intrinsics-VIMA routine for the creation phase of the bloom filter application.

```
_vim2K_ilmku (&entries[i], mask_k, key);
_vim2K_ilmku (&entries[i], mask_1, key);
_vim2K_irmku (fun, mask_1);
for (int j = 0; j < functions; j++){
   _vim2K_icpyu (key, bit);
   _vim2K_ipmtu (factors, fun, fac);
   _vim2K_ipmtu (shift_m, fun, shift_vec);
   _vim2K_imulu (bit, fac, bit);
   _vim2K_isllu (bit, shift_vec, bit);
   _vim2K_imodu (bit, bloom_filter_size, bit);
   _vim2K_isrlu (bit, shift5_vec, bit_div);
   _vim2K_iandu (bit, mask_31, bit_mod);
   _vim2K_isllu (mask_1, bit_mod, bit);
   _vim2K_iscou (bit, bit_div, bloom_filter);
   _vim2K_iaddu (fun, mask_1, fun);
}
```

Code C.8: Intrinsics-VIMA routine for the probing phase of the bloom filter application.

```
_vim2K_ilmku (&entries[i], mask_k, key); //load new entries according to the mask.
do {
   i += j;
   _vim2K_irmku (fun, mask_k);
   _vim2K_icpyu (key, bit);
   _vim2K_ipmtu (factors, fun, fac);
   _vim2K_ipmtu (shift_m, fun, shift_vec);
   _vim2K_imulu (bit, fac, bit);
   _vim2K_isllu (bit, shift_vec, bit);
   _vim2K_imodu (bit, bloom_filter_size, bit);
   _vim2K_isrlu (bit, shift5_vec, bit_div);
   _vim2K_iandu (bit, mask_31, bit_mod);
   _vim2K_isllu (mask_1, bit_mod, bit);
   _vim2K_igtru (bloom_filter, bit_div, bit_div);
   _vim2K_iandu (bit, bit_div, bit);
   _vim2K_icmqu (bit, mask_0, mask_k);
   _vim2K_icmqu (fun, fun_max, mask_kk);

   _vim2K_idptu (mask_kk, &j);
   if (j > 0) {
      _vim2K_ismku (key, mask_kk, &output[*output_count]);
      *output_count += j;
   }

   _vim2K_iorun (mask_k, mask_kk, mask_k);
   _vim2K_idptu (mask_k, &j);
   _vim2K_iaddu (fun, mask_1, fun);
} while (i < VECTOR_SIZE);
```