Efficient Machine Learning Execution with Near-Data Processing★

Aline S. Cordeiro[a,*], Sairo R. dos Santos[a,b], Francis B. Moreira[a], Paulo C. Santos[c], Luigi Carro[c] and Marco A. Z. Alves[a]

[a] Federal University of Paraná, Department of Informatics, Curitiba-PR, Brazil
[b] Federal Rural University of Semi-arid, Department of Exact Sciences and Information Technology, Angicos-RN, Brazil
[c] Federal University of Rio Grande do Sul, Informatics Institute, Porto Alegre-RS, Brazil

ARTICLE INFO

Keywords:
Near-Data Processing
Vector Processing
Machine Learning

ABSTRACT

A myriad of Machine Learning (ML) algorithms has emerged as a basis for many applications due to the facility in obtaining satisfactory solutions to a wide range of problems. Programs such as K-Nearest Neighbors (KNN), Multi-layer Perceptron (MLP), and Convolutional Neural Network (CNN) are commonly applied on Artificial Intelligence (AI) to process and analyze the ever-increasing amount of data. Nowadays, multi-core general-purpose systems are adopted due to their high processing capacity. However, energy consumption tends to scale together with the number of used cores. In order to achieve performance and energy efficiency, many Near-Data Processing (NDP) architectures have been proposed mainly tackling data movement reduction by placing computing units as close as possible to the data, such as specific accelerators, full-stack General Purpose Processor (GPP) and Graphics Processing Unit (GPU), and vector units. In this work, we present the benefits of exploring Vector-In-Memory Architecture (VIMA), a general-purpose vector-based NDP architecture for varied ML algorithms. Our work directly compares VIMA and x86 multi-core AVX-512 GPP. The presented approach can overcome the x86 single-core in up to 11.3× while improving energy efficiency by up to 8×. Also, our study shows that nearly 16 cores are necessary to match the NDP's single-thread performance for KNN and MLP algorithms, while it is necessary 32 cores for convolution. Nevertheless, VIMA still overcomes x86 32 cores by 2.1× on average when considering energy results.

1. Introduction

Machine Learning (ML) studies emerged in the late 1980s focusing on learning algorithms enabled by digital computers [67]. Due to the increased processing capacity available in current computers, ML has gained popularity and became the nucleus of many modern applications to process massive amounts of data generated by digital systems' growth usage [38, 65, 27, 61, 18].

Meanwhile, modern general-purpose computing systems have evolved to many processing cores, implementing deep cache memory hierarchies, high operating frequency, and Single Instruction Multiple Data (SIMD) units. Despite their ever-increasing performance, General Purpose Processors (GPPs) still present performance and energy efficiency limitations in several applications and workloads. Applications that demand transferring large amounts of data and streaming-like behavior (i.e., ML applications) primarily increase the memory-wall problem [79]. Thus, these applications increase performance differences between the processing unit (where they consume data) and main memory (where data resides). The bandwidth between processing units and main memory has physical limits due to the power dissipation, bus and pin-out width, and operating frequency.

On the other hand, systems use accelerators such as Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), and Application-Specific Integrated Circuits (ASICs) to mitigate these performance issues [79, 56, 37]. However, the memory-wall limitation is inherent to contemporary processor designs. Although modern cache hierarchies and high bandwidth memories can mitigate the performance drawbacks, in terms of energy and latency, it is not sufficient [29, 64, 63]. Moreover, this problem is of critical concern, as studies indicate that data movement accounts for up to 60% of the total system energy consumption of modern systems [9].

Considering execution time and energy constraints, Near-Data Processing (NDP) has emerged as a viable solution for the memory-wall problem, with the idea of integrating processing units as close as possible to the memory [55, 59]. NDP significantly reduces data movement between processing units and memory, which consequently reduces energy consumption. NDP takes advantage of accessing the vast internally available memory bandwidth, which allows processing over large amounts, therefore improving performance. Another potential of NDP is to overcome multi-threading capability, once it is possible to achieve significantly higher performance with one thread only if compared with conventional systems.

The majority of the NDP proposals rely on ASIC or full Central Processing Units (CPUs) and GPUs [26, 2, 54]. However, it is possible to enable a higher energy efficiency with designs based on simple near-data vector units [3, 69, 57, 68] while considering constraints regarding the area and

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

* Corresponding author

ascordeiro@inf.ufpr.br (A.S. Cordeiro); sairo.santos@ufersa.edu.br (S.R.d. Santos); fbm@inf.ufpr.br (F.B. Moreira); pcssjunior@inf.ufrgs.br (P.C. Santos); carro@inf.ufrgs.br (L. Carro); mazalves@inf.ufpr.br (M.A.Z. Alves)

ORCID(s): 0000-0002-8612-5749 (A.S. Cordeiro); 0000-0001-9981-5231 (S.R.d. Santos); 0000-0002-0926-3865 (F.B. Moreira); 0000-0001-8555-2637 (P.C. Santos); 0000-0002-7402-4780 (L. Carro); 0000-0003-2440-2664 (M.A.Z. Alves)

power [43]. In this manner, we based our study on HMC Instruction Vector Extensions (HIVE) [3], which provides a programming and simulation environment for NDP vector-based architectures.

In this paper, we extend our previous published work [15], where we presented the benefits of migrating three well-known ML kernels (K-Nearest Neighbors (KNN), Multi-layer Perceptron (MLP), and Convolutional Neural Network (CNN)) to Vector-In-Memory Architecture (VIMA). This NDP design enabled large-vector operations, which allows a significant reduction in data movement between the host processor and main memory, increasing overall energy efficiency and performance. To execute these evaluations with VIMA, we also developed Intrinsics-VIMA, a vector-designed C/C++ library extension [14]. Our library facilitates the simulation and evaluation of algorithms in VIMA and similar Processing-In-Memory (PIM) architectures with reduced programming effort.

As our main contribution, we provide insights and show benefits (in terms of execution time and energy results) when migrating ML algorithms to a vector-based NDP architecture compared to traditional x86 approaches using single- and multi-threading programming. This comparison varying the number of threads helps us to understand the NDP efficiency trade-off. We also evaluate the multi-thread migrations in terms of data throughput to better understand the correlation between memory footprint and computational performance.

Most ML algorithms have the training and inference phases, which are two computation-intensive tasks. The training step is performed once and is bound by latency as it executes many operations over a massive set of training instances to define the model parameters. The inference is performed multiple times by multiple products. It relies on high throughput to classify a stream of instances, representing real-world applications. In this paper, we focus only on the inference phase since we are interested in evaluating the computational performance of VIMA to classify a set of instances straightforward as if the model training phase was over.

Comparing the x86-only to the NDP, both with single thread execution, we show improvements on execution time of up to 10× for KNN, 11× for MLP, and 3× for convolution. Additionally, we reduce energy consumption by up to 7× for KNN, ∼ 8× for MLP and 2× for convolution. When comparing the NDP single thread execution with x86-only multi-threaded approach, near-data performance is on average 4.3× better than x86-2-threads, 1.8× better than x86-4-threads, and 1.3× better than x86-8-threads. In addition, we observed that NDP with a single thread has a performance equivalent to more than 8 x86 threads, leading to the best Dynamic Random Access Memory (DRAM) memory throughput and energy efficiency in all the experiments, still considering a perfect Dynamic Voltage and Frequency Scaling (DVFS) for x86 architecture.

2. Related Work

The first 3D-stacked memory device made commercially available with integrated NDP capabilities was the Hybrid Memory Cube (HMC) [33, 35]. It integrates one memory controller per individual vault, and each controller has a set of Functional Units (FUs) that can execute simple operations over up to 16-byte operands. As a result, much of the research involving 3D-stacked memories have considered the HMC design and specifications. More recently, another device became commercially available, the HBM-PIM [39].

Several efforts achieve better performance by attaching processing cores to 3D-stacked architectures. Such is the case of NeuroStream [7] and Network Training Accelerator (NTX) [70], both composed of RISC-V cores with local cache, Direct Memory Access (DMA) and specific cores. The two platforms use vector instructions to run Deep Neural Networks (DNNs) and Deep Convolutional Neural Network (DCNN), respectively, and connect to the crossbar switch of the 3D-stacked memory. Aimed at Big data Machine Learning Analytics (BMLA) applications, the Millipede [75] also adds full processors to the logic layer of 3D-stacked memories. Tesseract [2] applies a similar approach to large-scale processing, integrating a single-issue in-order ARM core to a HMC device. Gao et al. [25] implement an architecture geared toward in-memory analytics frameworks with a combination of ARM cores with Translation Look-aside Buffers (TLBs) and virtual memory using a vault router for communication between processing cores. Another possible approach is to implement programmable ARM-based cores in the HMC logic layer [45] so that the programmer can offload some functions to these cores.

All of the mentioned research efforts add complex cores to their designs, including all elements commonly associated. Integrating a complex core with a pipeline, cache hierarchy, and local memory near data increases complexity and energy consumption while imposing large area and power dissipation requirements. The added complexity of mapping functions to specific cores also hinders usability.

MAssively Parallel Learning/Classification Engine (MAPLE) [48] applies two distinct modules to the overall operation: one to aid MapReduce applications and one to map kernels to NDP resources automatically. The architecture uses multiple cores with registers, vector FUs, and local storage to achieve high parallelism. Xu et al. [80] use Accelerated Processing Units (APUs) consisting of multiple CPU and GPU cores to parallelize CNNs. Since these works add multiple processing cores to the architecture, they can be considered onerous in energy consumption and total area.

Another approach is to develop NDP architectures where cores are attached to individual vaults of a 3D-stacked memory, using a data controller for communication [23]. For instance, TETRIS [26] and NeuralHMC [50] use Processing Elements (PEs) connected with Network-on-Chip (NoC) technology to accelerate Neural Networks (NNs). Other than the challenges many-core integration causes, this approach requires specific attention to data placement in each vault.

Table 1
Summary of correlated papers characteristics.

Paper Reference	Architecture Model				Application Migrated		
	Full Processor	Functional Units	Near-Cell	ASIC	Convolution or CNN	KNN	MLP
[10] 2010	×				×		
[80] 2015	×				×		
[25] 2015	×				×		×
[12] 2016			×		×		×
[71] 2016			×		×		
[6] 2017				×	×		
[57] 2017		×			×		
[26] 2017				×	×		
[42] 2017			×		×		
[11] 2017			×		×		
[73] 2017			×		×		
[45] 2018	×			×	×		
[7] 2018	×			×	×		
[70] 2018	×			×	×		
[40] 2018	×					×	
[20] 2018			×		×		
[16] 2018			×		×		
[72] 2018			×		×		
[24] 2018	×				×		
[16] 2018				×	×		
[78] 2019			×		×		
[81] 2019			×		×		
[17] 2019			×		×		
[34] 2019			×		×		
[50] 2019		×		×	×		
[19] 2020			×				×
[66] 2020			×		×		×
[47] 2020			×		×		
Our Proposal		**×**			**×**	**×**	**×**

Utilization is complex when data from separate vaults are involved in a computation.

Some researchers have proposed adding reconfigurable logic to 3D memories. For instance, Neuron In-Memory (NIM) [57] uses reconfigurable FUs, a register bank and a sequencer to simulate NN near-data. Another example is the work by de Lima et al. [44], which adjusts FUs on demand. These solutions focus only on specific applications and assign processing elements to specified vaults, thus requiring precise data allocation.

Lastly, another approach is to modify DRAM cells so they can carry out simple instructions, which is inexpensive [24, 16, 42, 17, 72, 74]. However, since processing elements are connected directly to memory arrays, these solutions can be complex, and programming them is prone to errors. Also, due to the limitations of the available Instruction Set Architecture (ISA), porting algorithms to such devices is non-trivial.

Table 1 summarizes a comprehensive list of related work regarding NDP applied to ML algorithms such as CNN, KNN and MLP. The table presents two main aspects: the architecture model and the applications migrated. Regarding the architecture, we could observe full-core integration. Instead of the full-processors integration, several proposals add only logical and arithmetic units near-data, enabling large vector processing. Other proposals consider attaching processing elements within the memory cells. Finally, several proposals integrate custom ASICs able to accelerate only specific applications.

In this paper, we used the VIMA architecture with its programming library to migrate ML programs to NDP. Although we used a specific vector architecture as a target, we believe that such results would be similar for other vector NDP approaches. Besides, we extend our evaluation comparing to x86 processors with single- and multi-threaded execution.

3. Background on Near Data Processing

Moore's law predicted that the number of transistors in computer systems doubles every 18 months -currently, every 24 months -, which usually results in improvement of the processing capacity [52, 53]. For decades, as memory and processing capacity were the main limiting factors in developing efficient computer programs, his prediction proved to be correct [30], with current issues like the memory wall [79] and dark silicon [22].

ML emerged as an Artificial Intelligence (AI) research field in this context, as researchers tried to devise artificially intelligent systems that could find patterns in data without human intervention. Several ML techniques have thus been developed, such as supervised, unsupervised, and reinforcement learning [28, 51].

Meanwhile, 3D integration technology sparked interest in Near-Data Processing (NDP), an idea last explored in the 1990s [59, 21] but that failed to gain traction because of technological limitations of the time. Recently, 3D-stacked memories became commercially available, with the HMC and the High Bandwidth Memory (HBM) being the most widely known examples [31, 33].

By stacking several layers of DRAMs using vertical Through-Silicon Via (TSV) interconnection [58], as de-

picted in Figure 1, these 3D-stacked memories achieve high bandwidth and better energy efficiency in comparison to typical Double Data Rate (DDR) memories [77, 5]. The chosen device, HMC, has 32 independent logical partitions. Each partition has an individual memory controller placed in an additional logic layer.

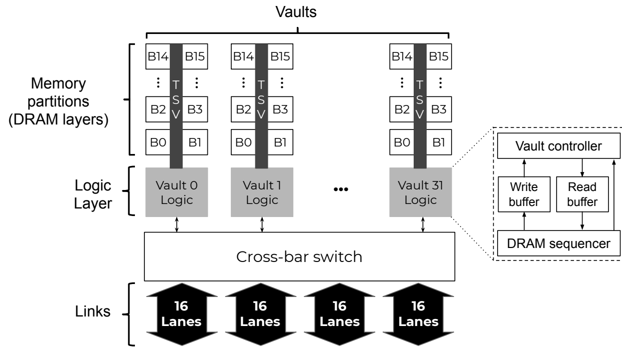


Figure 1: HMC block diagram with 32 vaults with 16 banks each one. Adapted from [32].

One common approach to implement NDP systems is to add processing elements to the logic layer in 3D-stacked memories, thus enabling processing with no need for data movement between memory and processor. Such systems are particularly interesting for data-driven applications that apply high pressure to memory. The most benefit is seen by applications with low data reuse and high parallelism, as they can take advantage of the high parallelism and bandwidth 3D-stacked devices offer [32, 36, 60].

In this paper we use VIMA which is based on HIVE [3] architecture, a general-purpose NDP architecture. HIVE leverages the high bandwidth and parallelism of 3D-stacked memories to enable the execution of vector instructions of extensive data loaded in parallel from the many independent logical vaults in the device. By adding its instructions to the host processor's ISA, it avoids implementing instruction fetching and decoding in the memory. Figure 2 shows VIMA which replaces the original register bank from HIVE with a dedicated cache memory to simplify its programming, adding transparency and flexibility. Both HIVE and VIMA support ARM NEON instructions and operate over vectors of 256 B or 8 KB in size.

The host processor handles all instruction front-end processing (e.g., fetching, decoding, renaming, and dispatching) and sees VIMA instructions as regular memory instructions. The memory requests bypass the cache hierarchy, and VIMA executes the instructions using vector FUs inside the DDR module. Upon completion, VIMA sends an instruction status back to the processor. For more details on near-data execution please refer to HIVE's paper [3].

NDP mitigates issues caused by data movement and, in comparison to traditional CPU and other accelerator systems, tends to be more efficient in time spent and energy consumption. For the remainder of this paper, we focus on migrating well-known ML classifications to NDP to exploit this approach.

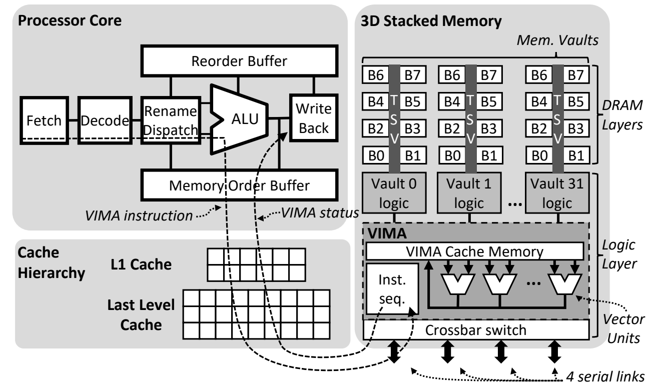


Figure 2: Processor connected to a 3D-stacked memory module with the VIMA architecture.

4. Programming for VIMA

In this paper we used Intrinsics-VIMA to evaluate the migration of ML code to VIMA. This *C/C++* library can be used to write VIMA code for simulation purposes. It works like Intel and ARM Intrinsics libraries [46], which embed code directly in the compiler. Thus, the resulting assembly code includes specific instructions from their extended ISA [13].

Intrinsics-VIMA functions achieve the same effect. When a programmer compiles code using such functions and uses it for simulation purposes, the library provides the illusion of VIMA instructions being part of the processor ISA. Intrinsics-VIMA functions appear explicitly in assembly code. When our trace generator, a tool implemented in our simulation infrastructure, detects Intrinsics-VIMA functions, it replaces them with VIMA instructions supported by the simulation software. All Intrinsics-VIMA code can be debugged and executed on any x86 architecture to make sure the algorithm is behaving as expected [14]. Codes 1 and 2 show examples of library function code and its use in application code, respectively.

Code 1: Intrinsics-VIMA routine example.

```
//This routine can be fully executed in a x86 architecture
//Our simulator replaces this routine with a VIMA instr.
void *_vim2K_fadds(__v32f *a, __v32f *b, __v32f *c) {
    for (int i = 0; i < vima_size; ++i) {
        c[i] = a[i] + b[i];
    }
    return EXIT_SUCCESS;
}
```

We wrote Intrinsics-VIMA on top of an open-source Intrinsics library which we used to evaluate the performance of applications using native HMC instructions [14]. Its interface exposes VIMA instructions with 256 B and 8 KB vector operands. Thus, vectors store 32 or 64 and 1024 or 2048 8 B or 4 B elements, respectively. Supported data types are integer and single or double-precision floating-point.

Our library assumes the size of a vector is a multiple of 32, 64, 1024, or 2048. The Intrinsics-VIMA library can iterate over vectors with these strides. Codes 1 and 2 show the implementation of an Intrinsics-VIMA routine and a vector

sum application, respectively. While previous work focused on scalar processing based on the HMC ISA [14], in this paper, we focus on the performance of vectorized near-data implementations using VIMA.

Code 2: 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]);
}
```

5. Migrating Machine Learning algorithms using Intrinsics-VIMA

ML analyzes data sets and recognizes patterns that enable them to make decisions and predictions regarding the domain represented in the data. The class of supervised algorithms has the training and inference phases, both uniquely challenging and computation-intensive. The training phase processes a large amount of representative data to generate a model. It iterates over data points to extract and adjust a model until it is considered adequately descriptive for a task. Since this phase iterates over massive data sets, it relies on robust architectures for performance. The inference phase classifies instances according to the model generated in the training phase, possibly in real-time, thus being more throughput-oriented. Models are created once and run in multiple devices, including embedded systems with limited resources [49, 62, 76]. Here, we focus on the inference phase.

Moreover, in this section, we describe how to write three widely used ML algorithms to use VIMA, including vectorization details. We choose a convolution kernel (commonly used in CNNs), MLP, and KNN and use 8 KB VIMA instructions to operate over 2048 single-precision values per instruction. While HIVE and VIMA use 8 KB vectors, both architectures can operate in a pipeline to operate using fewer vector FUs over smaller vectors and still offer high performance with low area usage [3].

The main idea here is to present our solutions when migrating these algorithms to VIMA architecture.

5.1. Convolution

Convolution operations are widely used in science and are explained first due to their simplicity. Its main loop consists of computing values according to a pattern involving neighboring data points in a 2D or 3D grid [1]. One of the most common patterns is the Von Neumann neighborhood, which considers the four direct neighbors of an element. Convolutions often suffer from inefficient memory use due to their poor locality of reference [1]. However, since each element is independent, the algorithm is highly parallel, making it a good candidate for NDP.

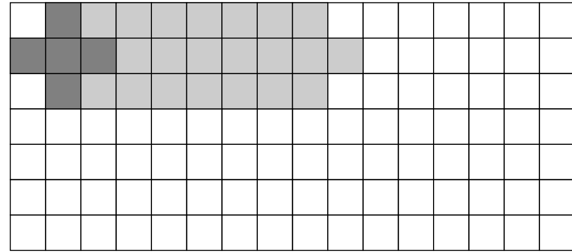


Figure 3: Convolution pattern used for VIMA.

We adopt a naive convolution with a Von Neumann pattern of range 1, as pictured in dark gray in Figure 3. Each loop loads and processes elements from three consecutive lines, as pictured in light gray.

Code 3: Von Neumann convolution code in C.

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

We consider a matrix stored in a continuous array, as shown in Code 3. The algorithm stores the results in a new matrix.

Code 4: Von Neumann convolution using Intrinsics-VIMA.

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

The corresponding Intrinsics-VIMA code is shown in Code 4 and ignores the borders of the matrix. It adds the five elements, multiplies the result by a constant, and stores results into a separate matrix.

5.2. K-nearest Neighbors

KNN classifies instances by searching the k minimal distances between a testing instance from the training ones. These instances are n-dimensional arrays. Thus, we can abstract them as points in an n-dimensional space. We must choose a method to calculate these distances depending on the distance criteria. Here, we are using the Euclidean Distance method, which considers the values of each array position since they correspond to the features that represent the instances. The higher the value, the more representative it is [51].

The KNN algorithm requires storing the training data in memory to classify the test instances. As the VIMA vectors

have static sizes, an instance can contain a smaller number of features when compared with a VIMA vector, as depicted in Figure 4, so different training instances can be stored consecutively in a VIMA vector. In other cases, a training instance can occupy at least one VIMA vector.

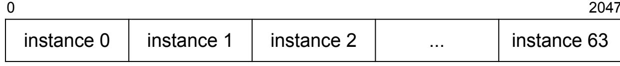


Figure 4: E.g., full utilization of a VIMA vector with training and test instances. Here, we could allocate 64 instances with 32 features inside the vector of 8 KB.

Reinforcing the importance of the VIMA vector sizes, we have to allocate enough memory to store the training instances while considering that this amount of memory must be multiple of the VIMA vector size. For example, if we have 8192 training instances with 32 features each one, it is necessary to split them in 4 VIMA vectors of 8 KB each, as depicted in Figure 5. In our case, as the focus of this paper is not achieving a better accuracy, we considered two generic classes as input - 0 for negative and 1 for positive - and we store these labels in a different array than the training set.

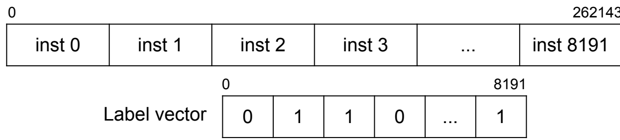


Figure 5: VIMA vectors with training instances with 32 features and the respective labels.

Consider the Euclidean Distance method, represented by the following function:

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

Where *tr* refers to the training instance and *te* to the test instance. With Intrinsics-VIMA routines, except for the square root operation, it is possible to vectorize the following steps presented in this method during its implementation:

- *_vim2K_fsubs()* to subtract the values between the training and test instances;
- *_vim2K_fmuls()* to raise the resulting value to the power of two;
- *_vim2K_fcums()* to accumulate sum the results to find out the distance between two instances.

Although a VIMA vector can allocate more than one instance when they are smaller than it, if we kept all these instances in a VIMA vector, we cannot simply execute all the mentioned vector operations. It is possible to subtract and multiply the values in correspondent instances. However, the accumulation sum routine will consider the features of more than one instance, which will lead to divergent results. Besides, this format will force us to allocate more memory to replicate the same training instance in one VIMA vector to correctly calculate its distance from different test instances. To avoid these complications, we created a single mask to isolate training and test instances in one single VIMA vector each. For example, as depicted in Figure 6, if we consider instances with 32 features, the mask will be of the size of a VIMA vector and will set the first 32 positions to "1," and fill the rest of the array with "0." There is no need for a mask when the instance is greater than the VIMA vector.

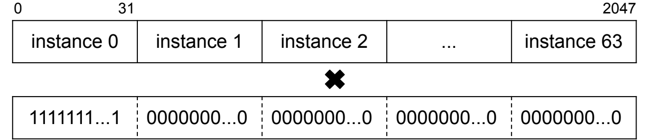


Figure 6: Operation to apply a mask over a VIMA vector of 8 KB with instances representing 32 features.

For simplicity, we chose to implement a single mask instead of different masks to ensure this mask vector will be allocated in VIMA's cache memory for a longer time. The presence of the mask in the cache contributes to better computational performance since VIMA's cache memory only has 64 KB with eight available cache lines of 8 KB each one. Thus, our code uses the minimum amount of VIMA vectors per operation. Moreover, we can easily apply this mask to the training and test sets by iterating the instance arrays to be operated with the mask vector.

After calculating each training and test instance, we store all the accumulated sums in different arrays. Each array represents one test instance, and each position represents a distance from the array's test instance to all the training instances. Finally, these arrays are operated with the x86 square root instruction, resulting in the Euclidean Distances, which are paired with the label vector to find the k lowest distance values since they represent the minimum distance between a test and training instance. This final step does not use Intrinsics-VIMA routines.

5.3. Multi-layer Perceptron

The MLP algorithm is a supervised learning technique that emulates the theoretical behavior of the human brain by similarly linking "neuron" objects. More specifically, a MLP combines a series of perceptrons or neurons in layers to compute its activation values and adjust its weights, applying non-linear transformations and focusing on finding an appropriate combination of parameters, which can lead to higher accuracy. This algorithm arranges these neurons in one input layer, one or multiple hidden layers, and one output layer to train and classify instances.

The number of neurons in the input layer is the number of features each instance contains. Meanwhile, the number of neurons in the hidden layer needs more attention since it must identify the most relevant input features in the model. If this layer contains too few or too many neuron units, it may not correctly identify and correlate the model input features,

leading to inaccurate results. Thus, we define this layer as half of the input layer. Finally, as depicted in Figure 7 the output layer has only two neurons to classify instances between positive and negative.

As in the KNN algorithm, we select VIMA vectors of 8 KB and floating-point single precision, which gives us vectors with 2048 positions. Besides, we consider only the inference stage of the MLP algorithm using a trained model as input. We disregard any training or classification parameters since our objective is evaluating the computational performance instead of accurate neural network results.

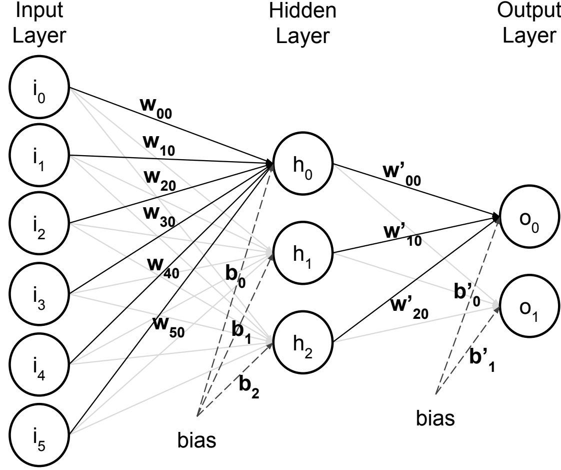


Figure 7: Representation of an Artificial Neural Network (ANN).

We use the same strategy applied to the KNN algorithm. If the number of features of an instance is less than the VIMA vector size, we apply a mask vector to isolate one test instance per vector. Otherwise, at least one VIMA vector will be enough for a test instance, and no transformation will be necessary.

To classify an instance, first, we have to operate a dot product between the features in the input layer and its connection weights values (w_{xy}) to obtain the activation values from the hidden layer. Second, we must repeat this process between the hidden layer activation and the connection weights values (w'_{xy}) to obtain the output activation values. In both steps, we can use two vector routines from VIMA as follows:

- *_vim2K_fmuls()* to multiply the activation values and its weights;
- *_vim2K_fcums()* to accumulate the values from multiplication, resulting in one activation value from the next layer.

In both processing stages, after calculating all the activation values of a layer, a bias - a correction factor - is added or subtracted to each activation value to adjust it and reduce errors. Finally, the activation function is applied to the values. In our case, we use the Rectified Linear Unit (ReLU) activation function on the hidden layer to introducing non-linearity to the instances, and the Softmax activation function on the output layer to transform the final activation values in probabilities [8]. Except for Softmax calculation, we use Intrinsics-VIMA routines to execute these operations, as follows:

- *_vim2K_fadds()* to add the bias vector to the hidden and output layers;
- *_vim2K_fmaxs()* to apply ReLU in the hidden layer. Here the hidden layer vector is operated with a zeroed vector, and the max operation then replaces every negative value by zero.

As we can notice, the operations between the layers are the same. However, we must pay attention to the varying number of weights for each layer since this factor defines the mask size to isolate the instances in VIMA vectors, as depicted in Figure 8.

0	7	15	23	31	2047
weight set 0	weight set 1	weight set 2	weight set 3	XXXXX...X	
✖					
11111111	00000000	00000000	00000000	00000000	
=					
weight set 0	00000000	00000000	...	00000000	
✖					
instance 0	00000000	00000000	...	00000000	

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

As mentioned, we are considering a binary label, so a result is positive or negative, i.e., the output layer contains only two neurons. Thus, two sets of weights (w'_{xy}) are defined, both sets with the same size as the hidden layer and referring to the connections between the hidden layer and the output layers. After applying the Softmax activation function, the higher probability will correspond to the label most likely to classify the instance. However, in this final step, we did not use any Intrinsics-VIMA routine.

6. Experimental Evaluation of VIMA

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

6.1. Methodology and Simulation Setup

Due to the high cost of prototyping, research in computer architecture often relies on simulation to validate ideas. Simulators enable researchers to represent the high complexity of computer systems with more accuracy than analytical models. For our experiments, we use an in-house simulator based on SiNUCA [4], an open-source cycle-accurate simulator. We used it to model our custom architecture and analyze its behavior when simulating selected benchmarks. Table 2 shows the parameters used for the simulation as well as the power consumption for each sub-system.

Table 2
Baseline and VIMA system configuration.

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

x86 baseline: We configured our baseline architecture to be similar to the Intel Sandy Bridge microarchitecture, henceforth referred to as x86.
VIMA architecture: The NDP scenario for comparison uses operations over 8 KB vectors so that VIMA uses all the 32 memory vaults simultaneously. VIMA instructions implement the NEON ISA near-data and are triggered and executed by the host processor alongside x86 instructions.
Benchmark: For comparison, we run the KNN, MLP, and convolution kernels in baseline, multi-threading, and VIMA. For the **x86 single-thread comparison with VIMA single-thread** we consider 4096 instances for MLP, 65536 training instances, 256 test instances, and 9 neighbors for KNN, varying the number of features for both applications (32, 64, 128, 256, 512, 1024, 2048, and 4096). Comparing **x86 multi-threaded version with VIMA single-thread**, we considered only the highest number of features and size tested. In this case, 4096 for KNN and MLP benchmarks, and a matrix size of 11648×11648 for the convolution benchmark. For all cases, we varied the number of threads between 1, 2, 4, 8, 16, and 32.

Our evaluations focus on architecture efficiency when migrating ML algorithms. Thus, our evaluations adopts three metrics: speedup, energy savings, and data throughput.

We use CACTI and Multicore Power, Area, and Timing (McPAT) tools to estimate energy consumption, as is customary in other related work [25, 26, 45]. We considered both in cost estimates on power, area, and timing parameters according to circuitry characteristics [41]. Besides, we considered a perfect DVFS in x86 energy calculations.

6.2. Execution Time Results - Single Threaded

For the convolution algorithm, described on Code 4, we have tested matrix sizes from 512×512 to 11648×11648. The speedup results are depicted in Figure 9(a), showing that the convolution is not linear since it depends on the vector fill rate and the x86 baseline implementation time, which varies depending on the usefulness of the cache memory. The code allocates 512 MB of memory for the greatest matrix size —11648 lines —and consequently each line has 44.2 KB. However, as the convolution operation allocates three matrix lines in cache memory to calculate one matrix line only, it results in 132.6 KB, which allows a fair usage of the x86 cache hierarchy.

Meanwhile, MLP and KNN algorithms presented better speedup results for VIMA, as depicted in Figure 9(b). This speedup happens because both algorithms exceed the Last-Level Cache (LLC) size by consuming 32 MB of memory with 512 and 256 features, respectively. When this occurs, the Advanced Vector Extensions (AVX) implementation becomes more expensive if compared with VIMA due to the need for frequent cache line replacements. At the same time, it is possible to observe that when the memory allocation stays within the LLC size, this behavior does not occur, resulting in a VIMA slow down over the baseline, as represented in MLP algorithm with up to 256 features and KNN with up to 128 features.

Comparing these two algorithms, we can notice that KNN exceeds the LLC capacity earlier - with 128 features - since it is a quadratic algorithm, while MLP is a linear algorithm, achieving a significant speedup later, with 4096 input features.

6.3. Energy Results - Single Threaded

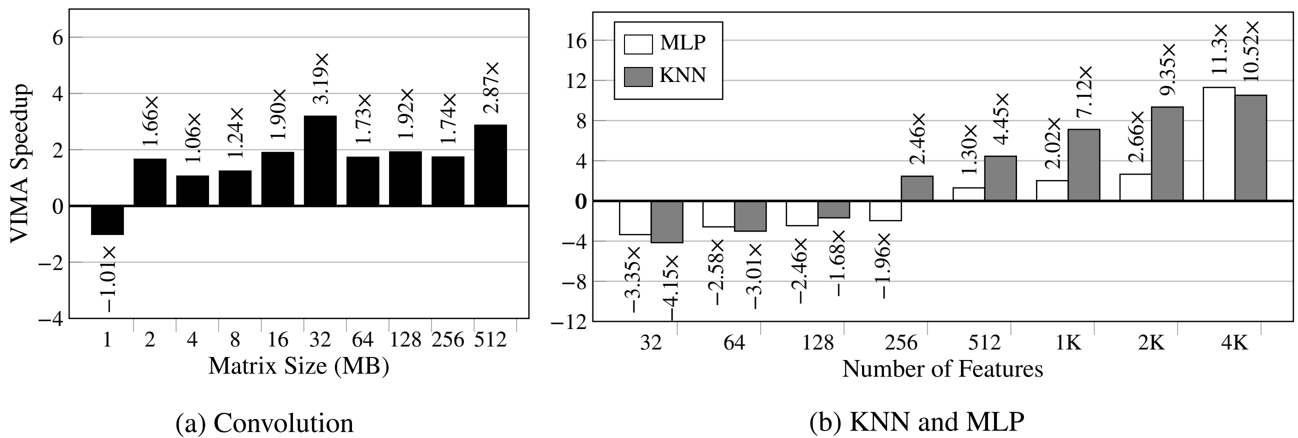
The energy efficiency results follow the speedup results pattern. As we can observe for the convolution, in Figure 10(a), the gains are higher when a matrix row fits perfectly into a VIMA vector, spending just half of the energy than the baseline.

For the MLP and KNN algorithms, the energy savings are proportional to the speedup, as depicted in Figure 10(b). On one side, we can notice an energy consumption reduction by 7× of VIMA over the baseline. On the other side, the energy savings for MLP and KNN with a lower number of features, i.e., until 512 and 128, respectively, is irrelevant since VIMA can consume up to 6× more energy than AVX in these cases.

In summary, the energy savings achieved by VIMA depend directly on memory usage and algorithm behavior. VIMA has impressive results for speedup and energy consumption with large datasets. However, when the memory footprint fits the x86 cache memory, a conventional processor presents higher efficiency. This result reinforces the concept of NDP as an accelerator for applications with datastream behavior and low data reuse.

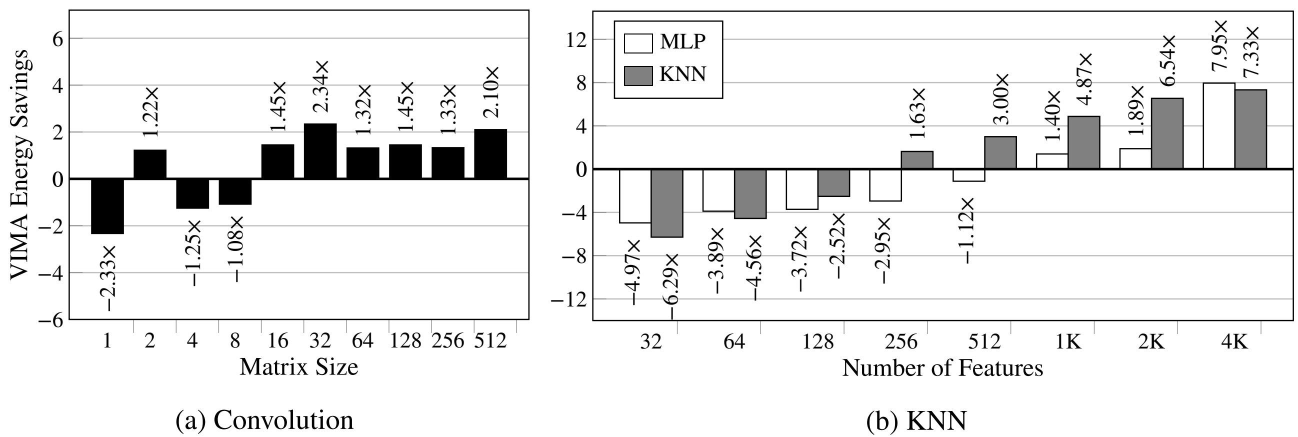
6.4. Execution Time and Energy Results - Multi-Threaded

Figure 11(a) presents the speedup results of VIMA over x86 multi-threaded for the KNN, MLP, and convolution algorithms. We can observe that x86 computational performance increases with the number of threads, especially for the MLP and KNN algorithms, in which VIMA presents a



(a) Convolution (b) KNN and MLP

Figure 9: VIMA's Speedup results over baseline for (a) Convolution varying matrix size, (b) MLP and KNN varying number of features. In both cases higher is better for VIMA.

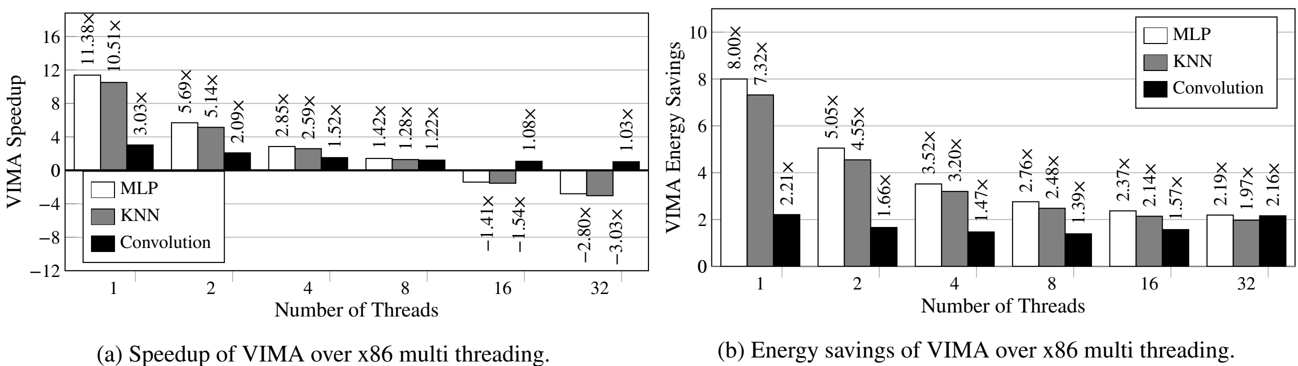


(a) Convolution (b) KNN

Figure 10: Energy savings of VIMA over baseline for (a) Convolution varying matrix size, (b) MLP and KNN varying number of features and neighbors. In both cases higher is better for VIMA.

slow down of around 3× for both. As we can observe in Figure 9, when we increase the number of threads VIMA's speedup over x86 decreases by half until it is almost null, with 8 threads.

Meanwhile, Figure 11(b) presents the energy-saving results of VIMA over x86 multi-threading for KNN, MLP, and convolution algorithms. We can observe that VIMA may not present a better performance in speedup for all the cases presented, but it stays more energy efficient. Even if compared to an x86 implementation with 32 threads VIMA spends up



(a) Speedup of VIMA over x86 multi threading. (b) Energy savings of VIMA over x86 multi threading.

Figure 11: Speedup and energy savings of VIMA over x86 multi threading. For both cases, we considered the execution of MLP with 4k instances and 4k features, kNN with 65k training instances, 256 test instances, and 4k features, and Convolution with a matrix of 512MB. In both cases higher is better for VIMA.

to 2× less energy with the same benchmark. These results perfectly illustrate the benefits of near-data processing using vector units. Our NDP implementation could, for most cases, achieve higher memory throughput, reducing off-chip data transfers while also requiring less complex units (such as processing cores).

6.5. Data Throughput Results - Multi-Threading

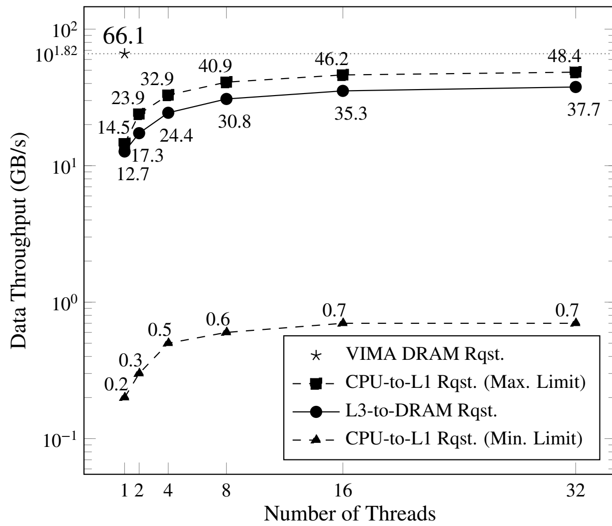


Figure 12: Accesses data per second in Convolution algorithm.

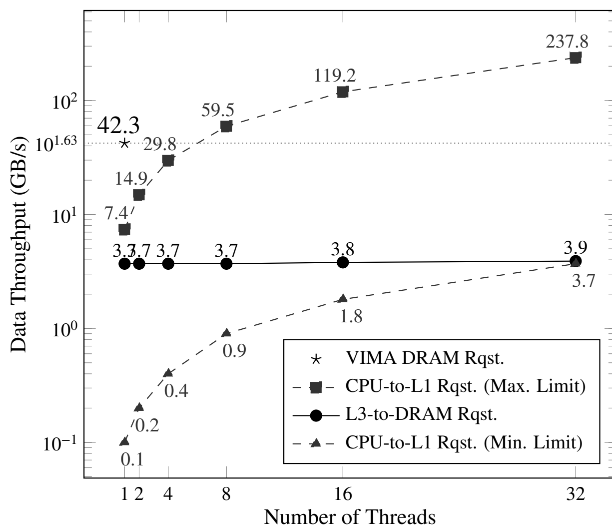


Figure 13: Accesses data per second in MLP algorithm.

Figures 12, 13, and 14 present data throughput (GB/s) during the algorithms' execution. Nowadays, x86 instructions can fetch from 1 byte to 64 bytes of data with a single request using scalar or AVX-512 instructions. Thus, when observing the number of CPU requests, the amount of data fetched can vary between two limits. The **CPU-to-L1 Rqst. (Min. Limit)** would refer to the total number of loads and stores from all the cores if they were all operations of 1-byte and the **CPU-to-L1 Rqst. (Max. Limit)** refers to the same

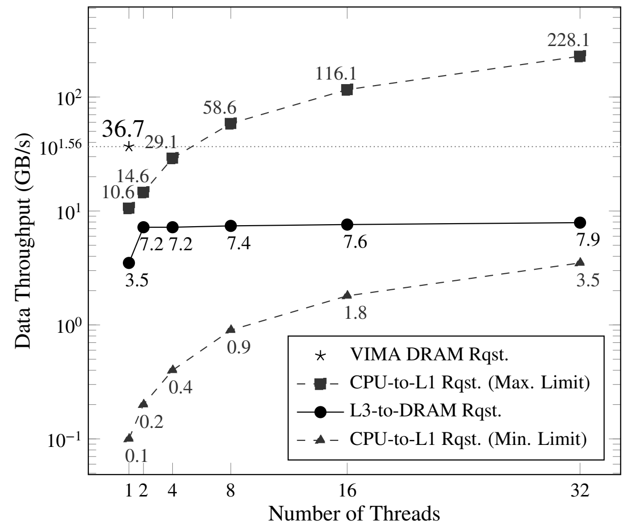


Figure 14: Accesses data per second in kNN algorithm.

number of loads and stores, but if they were all of the 64-bytes. Considering these two curves, we can affirm that actual CPU throughput will be within these two limits.

Furthermore, memory hierarchy can filter many requests avoiding DRAM fetches. **L3-to-DRAM Rqst.** represents the amount of data (GB/s) fetched from the main memory. In other words, load and store requests that are not filtered by the cache hierarchy turn into 64 byte cache line requests to DRAM. Finally, the **VIMA DRAM Rqst.** shown in the plot as a star (⋆) refers to the number of DRAM requests made when the code uses VIMA with a single thread.

The MLP and KNN results follow a similar pattern, which differs from the Convolution algorithm. For MLP and KNN, the cache hierarchy filters most of the memory requests. This conclusion is evident when we compare the results for 2 and 32 threads, in which the L3-to-DRAM throughput is closer to the CPU-to-L1 Rqst. (Max. Limit) requests when using only 2 threads. However, using 32 threads, the L3-to-DRAM throughput is maintained, while the CPU-to-L1 Rqst. (Max. Limit) is up to 58× higher. This observation means that all 32 threads frequently reuse data in the cache hierarchy.

For all the algorithms, we can observe that VIMA's DRAM throughput with a single thread is up to 11× higher than x86 single-thread. VIMA is capable of achieving such good performance because its instruction executes over 8 KB of data at once. Thus, it is typical for the algorithms during execution in VIMA to have a low rate of CPU requests to the cache memory hierarchy but have a significantly higher rate of requests made in main memory. We can observe that for MLP and KNN, the VIMA's throughput became slightly lower than x86 with 8 threads. Moreover, this difference increases faster for 16 and 32 threads leading to a VIMA's speedup decreases in these cases, as Figure 11(a) presents. For the Convolution algorithm, VIMA keeps a higher throughput rate if compared with all the x86 multi-thread cases, and this rate becomes closer when we increase

the number of threads. We can observe that the same pattern occurs in Figure 11(a).

Comparing these throughput results with the VIMA's speedup and energy savings, we can understand why the MLP and KNN algorithms have better performance and are more energy-efficient when executed with x86+AVX-512 with a higher number of threads. As Figures 13 and 14 show, in these cases, the data reuse in cache hierarchy increases together with the number of threads, reducing the number of memory requests (notice that cache accesses consumes only a fraction of energy and time compared to DRAM). For the Convolution algorithm, we see that its speedup and energy-saving stats remain similar when varying the number of threads, and the same occurs with the number of requests per second.

7. Conclusions and Final Considerations

In the last few years, several approaches to NDP have emerged to overcome the memory wall problem. Meanwhile, the number of applications requiring ML solutions has greatly increased.

In this paper, we presented an extension of our previous work [15] published on PDP'21 where we proposed the migration of ML kernels to VIMA, a vector execution near-data system, to achieve high speedup with low energy consumption. As extensions, we further discussed the efficiency of NDP by including comparisons with multi-threading implementation executing in x86. We also present a throughput evaluation to detail multi-threading performance.

We compared VIMA single thread, representing a generic solution for vector NDP, against x86 using AVX-512 multi-thread. Our ML migration allows the case study NDP to perform up to 11× better than x86 single thread. Meanwhile, VIMA equates x86 with 8 cores. Regarding energy results, the NDP approach consumes less energy than x86 in all the test cases. For the highest number of cores for x86, VIMA's consumption is lower by 2.1× on average.

These results show that our approach allows to efficiently use the vector NDP architecture's resources for ML algorithms. Nonetheless, we expect any program to benefit from VIMA if it relies on stream-based, contiguous data access behavior, low data reuse, and a memory consumption larger than the cache memory hierarchy capacity.

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

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

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

References

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

[2] Ahn, J., Hong, S., Yoo, S., Mutlu, O., Choi, K., 2016. A scalable processing-in-memory accelerator for parallel graph processing. ACM SIGARCH Computer Architecture News 43.

[3] Alves, M.A., Diener, M., Santos, P.C., Carro, L., 2016. Large vector extensions inside the hmc, in: Design, Automation & Test in Europe Conf. & Exhibition (DATE).

[4] Alves, M.A.Z., Villavieja, C., Diener, M., Moreira, F.B., Navaux, P.O.A., 2015. Sinuca: A validated micro-architecture simulator., in: HPCC/CSS/ICESS, pp. 605–610.

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

[6] 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 .

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

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

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

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

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

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

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

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

[15] Cordeiro, A.S., dos Santos, S.R., Moreira, F.B., Santos, P.C., Carro, L., Alves, M.A., 2021. Machine learning migration for efficient near-data processing, in: 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), IEEE. pp. 212–219.

[16] Deng, Q., Jiang, L., Zhang, Y., Zhang, M., Yang, J., 2018. Dracc: a dram based accelerator for accurate cnn inference, in: Design Automation Conf. (DAC).

[17] Deng, Q., Zhang, Y., Zhang, M., Yang, J., 2019. Lacc: Exploiting lookup table-based fast and accurate vector multiplication in dram-based cnn accelerator, in: Design Automation Conf. (DAC).

[18] Dietterich, T.G., 2000. Ensemble methods in machine learning, in: Int. workshop on multiple classifier systems.

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

[20] Eckert, C., Wang, X., Wang, J., Subramaniyan, A., Iyer, R., Sylvester, D., Blaaauw, D., Das, R., 2018. Neural cache: Bit-serial in-cache acceleration of deep neural networks, in: Int. Symp. on Computer

Architecture (ISCA). doi:10.1109/ISCA.2018.00040.

[21] Elliott, D.G., Stumm, M., Snelgrove, W.M., Cojocaru, C., McKenzie, R., 1999. Computational ram: Implementing processors in memory. IEEE Design & Test of Computers 16.

[22] Esmaeilzadeh, H., Blem, E., Amant, R.S., Sankaralingam, K., Burger, D., 2011. Dark silicon and the end of multicore scaling, in: Int. Symp. on computer architecture (ISCA).

[23] Ganguly, A., Singh, V., Muralidhar, R., Fujita, M., 2018. Memory-system requirements for convolutional neural networks, in: Proceedings of the Int. Symp. on Memory Systems.

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

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

[26] Gao, M., Pu, J., Yang, X., Horowitz, M., Kozyrakis, C., 2017. Tetris: Scalable and efficient neural network acceleration with 3d memory. ACM SIGOPS Operating Systems Review 51.

[27] Gardner, M.W., Dorling, S., 1998. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. Atmospheric environment 32.

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

[29] 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).

[30] Hennessy, J.L., Patterson, D.A., 2014. Computer Organization and Design: The Hardware and Software Interface. volume 4. Elsevier.

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

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

[33] Hybrid Memory Cube Consortium, 2014. Hybrid memory cube specification 2.1. http://www.hybridmemorycube.org/.

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

[35] Jeddeloh, J., Keeth, B., 2012a. Hybrid memory cube new dram architecture increases density and performance, in: 2012 symposium on VLSI technology (VLSIT), IEEE. pp. 87–88.

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

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

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

[39] 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., et al., 2021. 25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2 tflops programmable computing unit using bank-level parallelism, for machine learning applications, in: 2021 IEEE International Solid-State Circuits Conference (ISSCC), IEEE. pp. 350–352.

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

[41] Li, S., Ahn, J.H., Strong, R.D., Brockman, J.B., Tullsen, D.M., 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, pp. 469–480.

[42] Li, S., Niu, D., Malladi, K.T., Zheng, H., Brennan, B., Xie, Y., 2017. Drisa: A dram-based reconfigurable in-situ accelerator, in: Int. Symp. on Microarchitecture.

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

[44] de Lima, J.P.C., Santos, P.C., de Moura, R.F., Alves, M.A., Beck, A.C., Carro, L., 2019. Exploiting reconfigurable vector processing for energy-efficient computation in 3d-stacked memories, in: Int. Symp. on Applied Reconfigurable Computing.

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

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

[47] Long, Y., Lee, E., Kim, D., 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).

[48] Majumdar, A., Cadambi, S., Becchi, M., Chakradhar, S.T., Graf, H.P., 2012. A massively parallel, energy efficient programmable accelerator for learning and classification. ACM Transactions on Architecture and Code Optimization (TACO) 9, 1–30.

[49] McDanel, B., Teerapittayanon, S., Kung, H., 2017. Embedded binarized neural networks. arXiv preprint arXiv:1709.02260 .

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

[51] Mitchell, T.M., Learning, M., 1997. Mcgraw-hill science. Engineering/Math .

[52] Moore, G.E., 1998. Cramming more components onto integrated circuits. Proceedings of the IEEE 86, 82–85.

[53] Moore, G.E., et al., 1975. Progress in digital integrated electronics, in: Electron Devices Meeting, pp. 11–13.

[54] 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.

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

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

[57] Oliveira, G.F., Santos, P.C., Alves, M.A., Carro, L., 2017. Nim: An hmc-based machine for neuron computation, in: Int. Symp. on Applied Reconfigurable Computing.

[58] 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.

[59] Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., Yelick, K., 1997. A case for intelligent ram. IEEE micro 17.

[60] Pawlowski, J., 2011. Hybrid memory cube (hmc). Hot Chips 23.

[61] Peterson, L.E., 2009. K-nearest neighbor. Scholarpedia 4.

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

[63] Qureshi, M.K., Jaleel, A., Patt, Y.N., Steely, S.C., Emer, J., 2007a. Adaptive insertion policies for high performance caching. ACM SIGARCH Computer Architecture News 35.

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

[65] Rakotomamonjy, A., 2003. Variable selection using svm-based criteria. Journal of machine learning research 3.

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

[67] Russell, S.J., Norvig, P., 2016. Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,.

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

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

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

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

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

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

[74] Sudarshan, C., Lappas, J., Ghaffar, M.M., Rybalkin, V., Weis, C., Jung, M., Wehn, N., 2019. An in-dram neural network processing engine, in: Int. Symp. on Circuits and Systems (ISCAS).

[75] 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).

[76] Tian, Y., Pei, K., Jana, S., Ray, B., 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars, in: Proceedings of the 40th international conference on software engineering, pp. 303–314.

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

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

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

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

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