# Machine Learning Migration for Efficient Near-Data Processing

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

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

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

## I. Introduction

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

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

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

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

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

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

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

## II. Related Work

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

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

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

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

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

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

TABLE I: Summary of correlated papers characteristics.

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

## III. Background on Near Data Processing

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

NDP systems can be implemented due to 3D integration technology by adding processing capabilities within the logic layer. Thereby, NDP can mitigate data movement between memory and processor because it enables processing in the same chip where data is stored. NDP architecture improves performance and energy efficiency as it grants high parallelism

and high bandwidth [45]–[47], ensuring low average latency even when there is high pressure in memory. Therefore, such architectures benefit streaming and parallel applications, with coalescent memory access patterns and low data reuse.

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

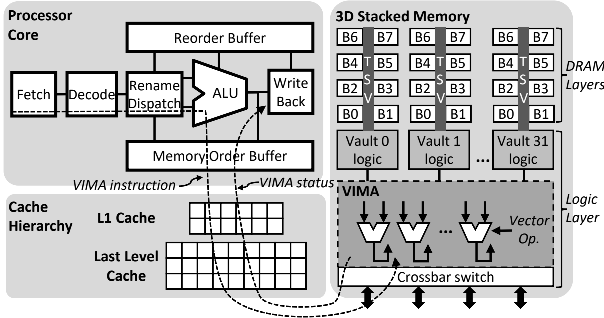



Fig. 1: 3D-stacked memory module with the VIMA architecture.

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

NDP can mitigate the memory-wall problem in contrast to CPU, GPU, and FPGA, which all require time and energy inefficient off-die (or off-chip) data transfers, by eliminating data movements from the memory hierarchy. Thus, in the remainder of this paper, we migrate three well-known ML classification algorithms to exploit this emerging architecture.

## IV. Proposed Intrinsics-VIMA Library

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

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

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

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

```
uint32_t vima_size = 2048;

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

// Initialize the memory location
<...>

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

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

Code 2: Intrinsics-VIMA routine example.

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

## V. Migrating Machine Learning using Intrinsics-VIMA

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

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

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

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

### A. Convolution

We start explaining the convolution code due to its simplicity, making it easy to understand the vector process. Convolution codes are a class of algorithms that has numerous applications in science. They compute values based on a fixed pattern involving each element of an array and several neighbors on a 2D or 3D arrangement [53]. Two of the most common convolution patterns are the Von Neumann neighborhood and the Moore neighborhood patterns. The Von Neumann pattern includes the four neighbors in the cardinal directions of an element. The computation of each element is independent, making convolution codes good candidates for parallel processing. However, they often become memory bottlenecks due to the data access patterns they present potentially having poor locality [53].

Code 3: Von Neumann convolution code in C.

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

For the implementation of a naive convolution code using VIMA, we adopted the Von Neumann pattern with a range equals to 1, as shown in dark gray in Figure 2. The algorithm sums all five elements in the convolution, then multiplies the result by a constant and stores the result in a different matrix. Code 3 shows an example in C, considering a matrix in a continuous array arrangement. The algorithm stores the result in the corresponding element of a new matrix.

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

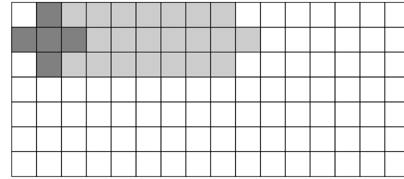


Fig. 2: Convolution pattern used for VIMA.

Code 4: Von Neumann convolution using Intrinsics-VIMA.

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

### B. K-nearest Neighbors

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

In the KNN algorithm, we must access the training data in memory to classify every test instance. Depending on the number of features an instance presents, it can be smaller than a VIMA vector, so different instances can be stored consecutively in one VIMA vector as depicted in Figure 3. Meanwhile, if the instance size is equal or larger than a VIMA vector, it will occupy at least one VIMA vector.

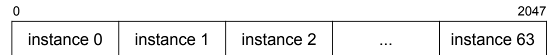



Fig. 3: E.g., full utilization of a VIMA vector with training and test instances. Here, we could allocate 64 instances with 32 features inside the vector of 8 KB.

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

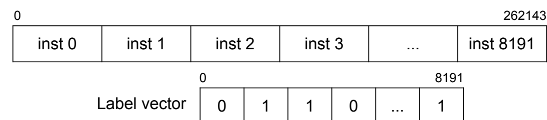



Fig. 4: VIMA vectors with training instances with 32 features and the respective labels.

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

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

Where *tr* refers to the training instance and *te* to the test instance. We use the following Intrinsics-VIMA routines: *_vim2K_fsubs()* to subtract the values of the training and test instances; *_vim2K_fmuls()* to multiply and raise the resulting value to the power of two; *_vim2K_fcums()* to sum all results to find out the distance between these instances and finally calculates the square root of this value. Fortunately, we can vectorize most of these operations with Intrinsics-VIMA.

Although a VIMA vector can receive more than one instance, depending on the number of features, we choose to work with a single instance at a time. To do so, we apply a mask in training and test vectors to obtain just a single instance. For instance, considering test and training instances with 32 features, the mask will set the first 32 positions of a VIMA vector to 1, while the rest of the vector is full of zeros, as depicted in Figure 5. If the instances size are equal or greater than the VIMA vector, this transformation will not be necessary. Isolating one instance per VIMA vector enables executing all the operations mentioned above (subtraction, multiplication and accumulated sum) in a simpler way with better data reuse.

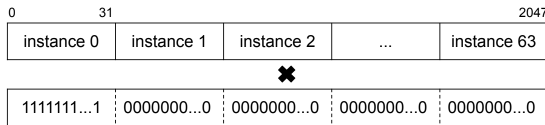



Fig. 5: Operation to apply a mask over a VIMA vector of 8 KB with instances representing 32 features.

We store the accumulated sums between each test instance and the set of training instances (calculated with VIMA routines) in a different vector in memory. Afterward, the x86 square root instruction will be applied, resulting in the Euclidean Distances. One vector for each test instance will store several Euclidean Distances. Each vector has size equals to the number of training instances. Finally, to classify an instance, all its distances are paired with the label vector to find the *k* lowest distances. In this phase, we are interested in the labels of the *k* lowest values. The label with the majority among these *k* lowest values is the label assigned to the test instance. This final step does not use Intrinsics-VIMA functions.

### C. Multi-layer Perceptron

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

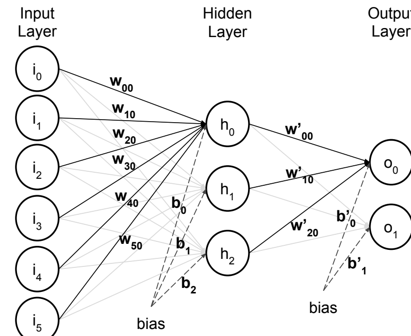



Fig. 6: Representation of an ANN.

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

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

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

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

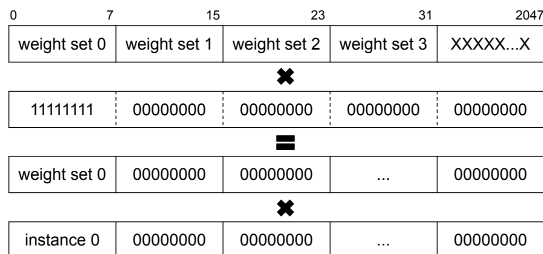


Fig. 7: Example of a VIMA vector with four sets of weights for instances representing 8 features.

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

## VI. Experimental Evaluation of VIMA

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

### A. Methodology and Simulation Setup

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

TABLE II: Baseline and VIMA system configuration.

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

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

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

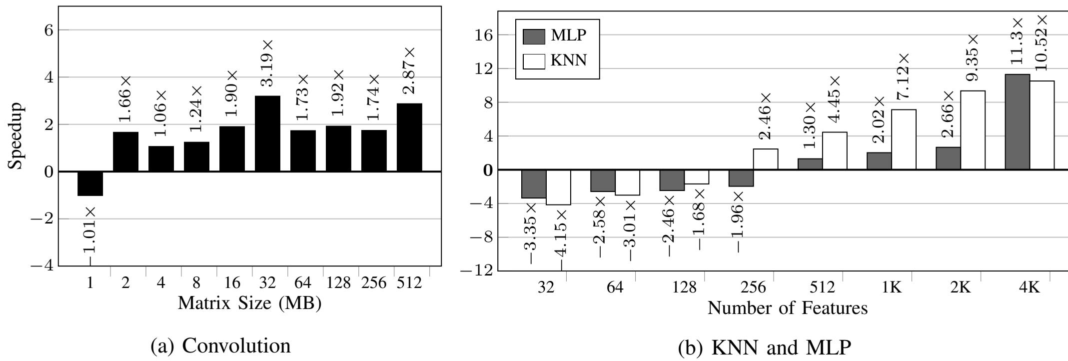
**Benchmark:** In our experiments, we evaluate KNN, MLP and convolution kernels. We used 4096 instances for MLP, 32768 training instances, 256 test instances, and 9 neighbors for KNN, varying the number of features for both applications (32, 64, 128, 256, 512, 1024, 2048, and 4096). For the convolution benchmark, we vary matrix dimensions ($512\times512$, $724\times724$, $1024 \times 1024$, $1448 \times 1448$, $2048 \times 2048$, $2896 \times 2896$, $4096\times4096$, $5794\times5794$, $8192\times8192$, and $11648\times11648$). Our evaluations focus on architecture efficiency, not on the accuracy of each classification algorithm. Thus, the results will be shown in terms of speedup and energy savings.

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

### B. Execution Time Results

Figure 8(a) presents speedup results for the convolution algorithm described on Code 4 over matrices from size $512\times512$ to $11648 \times 11648$. The speedup for the convolution is not linear. It depends on the vector fill rate and the x86 baseline implementation time, which varies whenever the cache is more or less useful. We evaluated with the larger matrix of $11648\times11648$ that occupies 512 MB of memory, which still makes fair usage of the cache hierarchy of x86. Nevertheless, sizes greater than 16 MB slightly better utilizes the VIMA vectors, achieving thus the maximum performance.

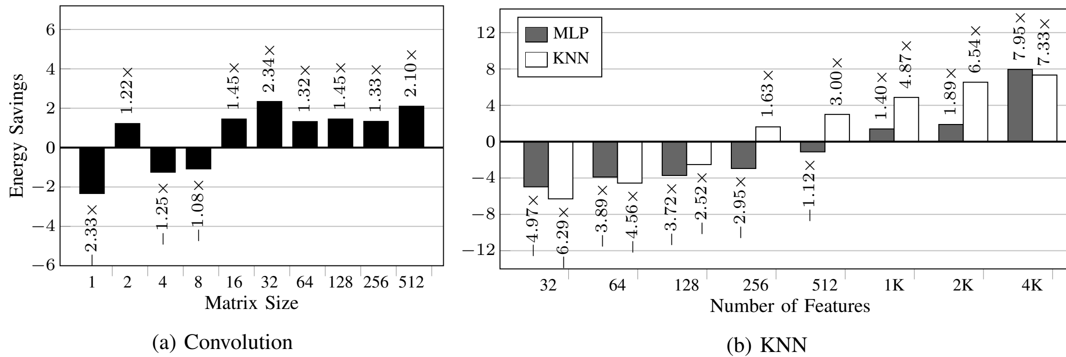
Figure 8(b) presents speedup results for MLP and KNN algorithms. Both algorithms start to present better results for VIMA when increasing memory usage. MLP and KNN exceed cache size with 512 and 256 features, respectively, using 32 MB of memory. When the memory footprint exceeds x86 cache memory size,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




(a) Convolution

(b) KNN and MLP

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




(a) Convolution

(b) KNN

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

### C. *Energy Results*

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

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

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

## VII. Conclusions and Final Considerations

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

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

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

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

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

## References


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

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

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

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

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


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

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

[6] A. Boroumand, S. Ghose *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.
[7] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, 1995.
[8] E. Nurvitadhi, J. Sim *et al.*, "Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–4.
[9] K. Kara, D. Alistarh *et al.*, "Fpga-accelerated dense linear machine learning: A precision-convergence trade-off," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017, pp. 160–167.
[10] M. Hashemi, E. Ebrahimi *et al.*, "Accelerating dependent cache misses with an enhanced memory controller," in *Int. Symp. on Computer Architecture (ISCA)*, 2016.
[11] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, "Line distillation: Increasing cache capacity by filtering unused words in cache lines," in *Int. Symp. on High Performance Computer Architecture (HPCA)*, 2007.
[12] M. K. Qureshi, A. Jaleel *et al.*, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, 2007.
[13] A. Nowatzyk, F. Pong, and A. Saulsbury, "Missing the memory wall: The case for processor/memory integration," in *Int. Symp. on Computer Architecture (ISCA)*, 1996.
[14] D. Patterson, T. Anderson *et al.*, "A case for intelligent ram," *IEEE micro*, vol. 17, no. 2, 1997.
[15] M. Gao, J. Pu *et al.*, "Tetris: Scalable and efficient neural network acceleration with 3d memory," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, 2017.
[16] J. Ahn, S. Hong *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, 2016.
[17] R. Nair, S. F. Antao *et al.*, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, 2015.
[18] M. A. Alves, M. Diener *et al.*, "Large vector extensions inside the hmc," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2016.
[19] P. C. Santos, G. F. Oliveira *et al.*, "Operand size reconfiguration for big data processing in memory," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2017.
[20] G. F. Oliveira, P. C. Santos *et al.*, "Nim: An hmc-based machine for neuron computation," in *Int. Symp. on Applied Reconfigurable Computing*, 2017.
[21] P. C. Santos, G. F. Oliveira *et al.*, "Processing in 3d memories to speed up operations on complex data structures," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*. IEEE, 2018.
[22] J. a. P. Lima, P. C. Santos *et al.*, "Design space exploration for pim architectures in 3d-stacked memories," in *Proceedings of the Computing Frontiers Conference*. ACM, 2018.
[23] A. S. Cordeiro, T. R. Kepe *et al.*, "Intrinsics-hmc: An automatic trace generator for simulations of processing-in-memory instructions," *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*, 2017.
[24] E. Azarkhish, D. Rossi *et al.*, "Neurostream: Scalable and energy efficient deep learning with smart memory cubes," *Trans. on Parallel & Distributed Systems*, 2018.
[25] F. Schuiki, M. Schaffner *et al.*, "A scalable near-memory architecture for training deep neural networks on large in-memory datasets," *arXiv preprint arXiv:1803.04783*, 2018.
[26] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Parallel Architecture and Compilation (PACT)*, 2015.
[27] M. Thottethodi, T. Vijaykumar *et al.*, "Millipede: Die-stacked memory optimizations for big data machine learning analytics," in *Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2018.
[28] J. Liu, H. Zhao *et al.*, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *Int. Symp. on Microarchitecture (MICRO)*, 2018.
[29] A. Ganguly, V. Singh *et al.*, "Memory-system requirements for convolutional neural networks," in *Proceedings of the Int. Symp. on Memory Systems*, 2018.
[30] C. Min, J. Mao *et al.*, "Neuralhmc: an efficient hmc-based accelerator for deep neural networks," in *Asia and South Pacific Design Automation Conf. (ASPDAC)*, 2019.
[31] S. Cadambi, A. Majumdar *et al.*, "A programmable parallel accelerator for learning and classification," in *Int. Conf. on Parallel architectures and Compilation Techniques (PACT)*, 2010.
[32] L. Xu, D. P. Zhang, and N. Jayasena, "Scaling deep learning on multiple in-memory processors," in *Workshop on Near-Data Processing*, 2015.
[33] J. P. C. de Lima, P. C. Santos *et al.*, "Exploiting reconfigurable vector processing for energy-efficient computation in 3d-stacked memories," in *Int. Symp. on Applied Reconfigurable Computing*, 2019.
[34] D. Gao, T. Shen, and C. Zhuo, "A design framework for processing-in-memory accelerator," in *Int. Workshop on System Level Interconnect Prediction (SLIP)*, 2018.
[35] Q. Deng, L. Jiang *et al.*, "Dracc: a dram based accelerator for accurate cnn inference," in *Design Automation Conf. (DAC)*, 2018.
[36] S. Li, D. Niu *et al.*, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Int. Symp. on Microarchitecture*, 2017.
[37] Q. Deng, Y. Zhang *et al.*, "Lacc: Exploiting lookup table-based fast and accurate vector multiplication in dram-based cnn accelerator," in *Design Automation Conf. (DAC)*, 2019.
[38] J. Sim, H. Seol, and L.-S. Kim, "Nid: processing binary convolutional neural network in commodity dram," in *Int. Conf. on Computer-Aided Design (ICCAD)*, 2018.
[39] C. Sudarshan, J. Lappas *et al.*, "An in-dram neural network processing engine," in *Int. Symp. on Circuits and Systems (ISCAS)*, 2019.
[40] D. G. Elliott, M. Stumm *et al.*, "Computational ram: Implementing processors in memory," *IEEE Design & Test of Computers*, vol. 16, 1999.
[41] J. V. Olmen, A. Mercha *et al.*, "3D stacked IC demonstration using a through silicon via first approach," in *Int. Electron Devices Meeting*, 2008.
[42] J. Hrusca, "PIM comparison," https://www.extremetech.com/computing/197720-beyond-ddr4-understand-the-differences-between-wide-io-hbm\-and-hybrid-memory-cube, 2015, [Online; accessed 01-July-2019].
[43] Transcend, "DDR comparison," https://www.transcend-info.com/Support/FAQ-296, 2014, [Online; accessed 01-July-2019].
[44] AMD, "DDR5 and HBM comparison," https://www.amd.com/system/files/documents/high-bandwidth-memory-hbm.pdf, 2015, [Online; accessed 01-July-2019].
[45] Hybrid Memory Cube Consortium, "Hybrid memory cube specification rev. 2.0," 2013, http://www.hybridmemorycube.org/.
[46] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Symp. on VLSI Technology*, 2012.
[47] J. Pawlowski, "Hybrid memory cube (hmc)," *Hot Chips*, vol. 23, 2011.
[48] C. Lomont, "Introduction to intel advanced vector extensions," *Intel White Paper*, 2011.
[49] I. Coorporation, "Intel 64 and ia-32 architectures optimization reference manual," 2009.
[50] B. McDanel, S. Teerapittayanon, and H. Kung, "Embedded binarized neural networks," *arXiv preprint arXiv:1709.02260*, 2017.
[51] J. Qiu, J. Wang *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35.
[52] Y. Tian, K. Pei *et al.*, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 303–314.
[53] S. Afonso, A. Acosta, and F. Almeida, "Automatic acceleration of stencil codes in android devices," in *Int. Conf. on Algorithms and Architectures for Parallel Processing*, 2017.
[54] T. M. Mitchell and M. Learning, "Mcgraw-hill science," *Engineering/Math*, 1997.
[55] C. M. Bishop *et al.*, *Neural networks for pattern recognition*. Oxford university press, 1995.
[56] M. A. Z. Alves, C. Villavieja *et al.*, "Sinuca: A validated micro-architecture simulator." in *HPCC/CSS/ICESS*, 2015, pp. 605–610.
[57] S. Li, J. H. Ahn *et al.*, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469–480.