multiPULPly: A Multiplication Engine for Accelerating Neural Networks on Ultra-Low-Power Architectures

ADI ELIHAU and RONNY RONEN, Technion - Israel Institute of Technology
PIERRE-EMMANUEL GAILLARDON, University of Utah
SHAHAR KVATINSKY, Technion - Israel Institute of Technology

Computationally-intensive neural network applications often need to run on resource-limited low-power devices. Numerous hardware accelerators have been developed to speed up the performance of neural network applications and reduce power consumption; however, most focus on data centers and full-fledged systems. Acceleration in ultra-low-power systems has been only partially addressed. In this paper, we present multiPULPly, an accelerator which integrates memristive technologies within standard low-power CMOS technology, to accelerate multiplication in neural network inference on ultra-low-power systems. This accelerator was designated for PULP, an open-source microcontroller system that uses low-power RISC-V processors. Memristors were integrated into the accelerator to enable power consumption only when the memory is active, to continue the task with no context-restoring overhead, and to enable highly-parallel analog multiplication. To reduce the energy consumption, we propose novel dataflows that handle common multiplication scenarios and are tailored for our architecture. The accelerator was tested on FPGA and achieved a peak energy efficiency of 19.5 TOPS/W, outperforming state-of-the-art accelerators by 1.5× to 4.5×.

CCS Concepts: • **Hardware → Emerging technologies**; **Non-volatile memory**; **Emerging architectures**; • **Computer systems organization → Neural networks**.

Additional Key Words and Phrases: memristor, ultra-low-power architectures, mobile neural networks

ACM Reference Format:
Adi Elihau, Ronny Ronen, Pierre-Emmanuel Gaillardon, and Shahar Kvatinsky. 2020. multiPULPly: A Multiplication Engine for Accelerating Neural Networks on Ultra-Low-Power Architectures. *ACM J. Emerg. Technol. Comput. Syst.* 1, 1, Article 1 (January 2020), 26 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Recent advances in deep neural network (DNN) research were made possible by the AlexNet architecture [42], with its 2012 breakthrough performance on the ImageNet dataset using DNNs. Since then, numerous other DNNs for image classification have been explored [21, 69], and other fields have adopted neural networks (NNs) to solve problems such as face recognition [70] and machine translation [17]. The emergence of large databases and the development of high-performance architectures have further contributed to this trend. The former enabled reliable training of DNNs

This work was partially supported by the European Research Council under the European Union's Horizon 2020 Research and Innovation Program (grant agreement no. 757259), by the Genpro consortium, Israel Innovation Authority, and by the US-Israel Binational Science Foundation (grant no. 2016016).
Authors' addresses: Adi Elihau, adieliahu@campus.technion.ac.il; Ronny Ronen, ronny.ronen@ef.technion.ac.il, Technion - Israel Institute of Technology, Haifa, Israel, 3200003; Pierre-Emmanuel Gaillardon, pierre-emmanuel.gaillardon@utah.edu, University of Utah, 201 Presidents Circle, Salt Lake City, Utah, 84112; Shahar Kvatinsky, shahar@ee.technion.ac.il, Technion - Israel Institute of Technology, Haifa, Israel, 3200003.

1550-4832/2020/1-ART1 $15.00
https://doi.org/10.1145/1122445.1122456

Table 1. Number of Multiply and Accumulate Operations in Selected Models.

Model	Million MACs
FaceNet [62]	1600
1.0 MobileNet-160 [32]	**286**
GoogLeNet [69]	1550
1.0 MobileNet-224 [32]	**569**
res15 [71]	894
EdgeSpeechNet-D [49]	**24.5**

and demonstrated low error rates, while the latter enabled sufficiently high computational power to perform training in a reasonable time.

NN applications perform poorly on general-purpose computers since processors are designed to execute computational tasks, but are not tailored to a specific calculation. Hence, graphics processing units (GPUs) are increasingly utilized for machine learning applications, thanks to their efficient matrix operations. In addition, hardware accelerators have been developed to perform this task with lower runtime and better energy efficiency using methods such as smart memory or optimized mathematical operations.

Emerging non-volatile memory (NVM) technologies, such as memristors, have gained popularity as building blocks in such accelerators, since they offer solutions to the challenges of massive data transfer and the large number of multiply-accumulate (MAC) operations required in DNNs [16, 57]. Memristors function as memory elements and have computational capabilities. Computations can be performed in-memory directly on the data, or near-memory using additional peripheral circuits, thereby eliminating the data transfer bottleneck. Furthermore, a vector-matrix multiplication (VMM) can be efficiently performed using a memristive crossbar [79].

While existing accelerators and GPUs may address the aforementioned problems in high-power architectures such as data centers, enhanced performance comes with significant power consumption [12, 34, 39]. Examples of such high-power accelerators are ISAAC [64] and Newton [53], both of which store all the network weights in memristive crossbar arrays. The crossbar arrays, together with additional periphery, are large and consume much energy.

Table 1 lists examples of different NN models, including mobile-dedicated models. Boldface indicates models for embedded and mobile applications. The models include: face recognition systems [32, 62]; image classification systems [32, 69]; and speech recognition networks [49, 71]. All of these models require a large number of MAC operations during the inference process. However, even in the mobile-dedicated models, which require fewer parameters, the number of MAC operations is still large, clearly illustrating the necessity for mobile-dedicated accelerators - the number of MAC operations which can be accelerated is relatively large, but low-power constraints prevent this in practice. Furthermore, the lower number of parameters can be utilized to develop accelerators that are tailored for low-power devices.

In this paper, we present multiPULPly, an NN inference accelerator designed for PULP (Parallel-Ultra-Low-Power) [60], an open-source microcontroller system that uses a low-power processor based on RISC-V cores [77] that operate at low voltage. PULP operates on the normally-off paradigm, *i.e.*, it is put in a low-power mode when the core is idle, and therefore maintains a power envelope of a few milliwatts. However, this paradigm requires use of NVM elements to avoid costly memory refresh or data loss. Thus, the addition of NVM elements is crucial for supporting data-heavy IoT (Internet of Things) applications.

multiPULPly uses memristor analog multiplication to accelerate common NN operations. As opposed to other accelerators which use numerous memristive crossbar arrays, multiPULPly uses a small number of crossbars and analog-to-digital converters (ADCs), due to power limitations.

Since the crossbar arrays cannot accommodate all the weights, data reuse and mapping methods suited to real-time low-power systems are applied. Efficient VMM has been widely investigated in the literature. However, the proposed data reuse concepts in this paper are specific to memristor-based analog multiplication, in which one vector at a time can be multiplied with a matrix. We also distinguish between multiplication with single and multiple vectors; the order of operations is different for the two scenarios. multiPULPly achieves an energy efficiency of 19.5 TOPS/W, outperforming the IoT state-of-art accelerators by 1.5× to 4.5×.

This paper makes the following contributions:

- Proposing multiPULPly, the first memristor-based ultra-low-power NN accelerator for PULP. The accelerator uses analog multiplication based on memristive arrays. The idea of integrating memristors in PULP was briefly mentioned earlier [59], but such architecture has never been designed. Our accelerator is especially efficient in data-heavy IoT applications, and by smart integration of different hardware blocks, achieves state-of-the-art energy efficiency.
- Development of novel dataflows in the absence of cache hierarchy, to minimize data transfer between the accelerator and the main memory. These dataflows are tailored for the memristive crossbar, maintaining simplicity by its non-volatility, and distinguishing between single and multiple vectors reused during a multiplication with a matrix, according to the specific NN layer.
- Contribution to the open-source hardware community. The multiPULPly code, which was tested on FPGA, is released and will supply an infrastructure for the development of additional ultra-low-power accelerators.

2 BACKGROUND

2.1 Mobile Neural Networks

Mobile NNs and DNNs have a similar structure. Among their various layers, they contain fully connected (FC) layers and convolutional (CONV) layers, which can be implemented using VMM or matrix-matrix multiplication (MMM). However, since mobile NNs run on IoT devices, which have a strict storage limitation, they are adapted to comply with this restriction, therefore differing from DNNs. The total size of the parameters in such networks is a few MB, at most. To satisfy this limitation, quantization is sometimes applied. While training today uses floating point representation, a quantization step transforms floating-point numbers into 16-bit or 8-bit integers, which are usually sufficient for inference [26, 31, 41] and save storage space, reduce energy, and speed up the computation. Layer-specific precision [38] and weight compression and pruning techniques [28, 29, 48] were also investigated. The number of layers and the number of parameters in mobile NNs have also been reduced. For example, in mobileNet [32], a parameter called *width multiplier* was introduced to thin a network uniformly at each layer by reducing the input and output channels. While the dimensions of matrices in common DNNs can reach hundreds of thousands of rows and columns [62], the dimensions in mobile NNs do not exceed several thousands [32, 36, 49], as discussed in Section 5.4. Other methods for reducing the computation, *e.g.*, depthwise separable convolutions [65] and weight repetition [30], have also been applied.

To improve performance and energy consumption, the CONV layers and the FC layers – often the most power and time-consuming layers in DNNs [13] – are accelerated. These layers are used in many IoT applications, *e.g.*, for face, speech, gesture and object recognition [66]. MobileNet [32] uses 27 CONV layers and 1 FC layer out of 30 layers, and EdgeSpeechNet-A [49] uses a total of 17 layers, 14 of which are CONV layers and 1 of which is a FC layer. Other layers, such as pooling and activation, might also be accelerated, as demonstrated in [37, 64].

2.2 PULP

Machine learning applications usually process different data streams generated by different sensors, and therefore require high computation performance. To address this, a new family of energy-efficient computation platforms has recently emerged [24, 55, 63]. A strong representative of this family is PULP [2, 60], an open-source hardware and software research and development platform targeted for IoT applications. As such, its goal is to reduce the consumed energy as well as meet the high computational demands of IoT applications. PULP contains a microcontroller system, a broad set of peripherals, and a multi-core platform, based on open-source RISC-V instruction-set architecture [77]. PULP uses a normally-off policy: when the core is idle, PULP is put into a low-power mode. In this mode, all the devices, with the exception of the event unit, are clock-gated. The event unit activates the cores when an event or an interrupt arrives.

A recent PULP release called PULPissimo, contains support for the addition of new accelerators to the system. The accelerators, called hardware processing engines (HWPEs), are memory-coupled accelerators dedicated to improving the energy efficiency or the performance of a specific calculation. As opposed to many other accelerators, HWPEs do not depend on globally-shared, external direct memory access (DMA) to import or export data, but operate on multi-banked scratchpad memory shared by all of the elements in the PULP system (often referred to as TCDM [tightly coupled data memory]) [19]. There is no address virtualization when accessing the TCDM. An HWPE contains three main parts: the streamer, the control, and the datapath. Parts of the streamer, the control and the datapath should be implemented in order to achieve the accelerator goal.

(1) **Streamer**: The streamer includes several internal DMA units that allow the accelerator to directly and independently access the main memory without CPU help. The streamer is responsible for transferring data into (using the store unit) and out of (using the load unit) the memory. The streamer includes an address generator module, which is used to generate addresses to load or store HWPE-Streams.
(2) **Control**: The HWPE control includes three main parts:
   (a) **Register File**: The register file contains user-defined non-contexted registers, and user-defined contexted registers, used to implement a queue of jobs that can be offloaded even while the HWPE is active. The software causes context-switching when it initiates a job. The register file also contains several mandatory registers, *e.g.,* the trigger and the status registers.
   (b) **Microcode Processor**: The proprietary microcode processor supports a small set of instructions used only for efficient address generation, defined by the user in a high-level fashion. This is done by letting the user define the microcode using the YAML format [3], which is pre-compiled to a binary code.
   (c) **FSM**: The Finite State Machine (FSM), implemented by the accelerator developer, is responsible for managing the streamer, the microcode processor and the engine, according to the input coming from the register file and the state of the machine.
(3) **Datapath (engine)**: The engine is the core of the HWPE, and it contains the actual datapath of the accelerator. The data comes from the streamers and is managed according to the control signals from the control modules. The engine should be developed by the developer to perform the computation task.

2.3 Emerging NVM Technologies

Memristors are two-terminal electrical devices that can function as both memory and computing element. One common memristor technology is resistive RAM (RRAM) [5, 7], in which two metal layers sandwich an oxide layer. The single-level cell memristor can change its resistance between

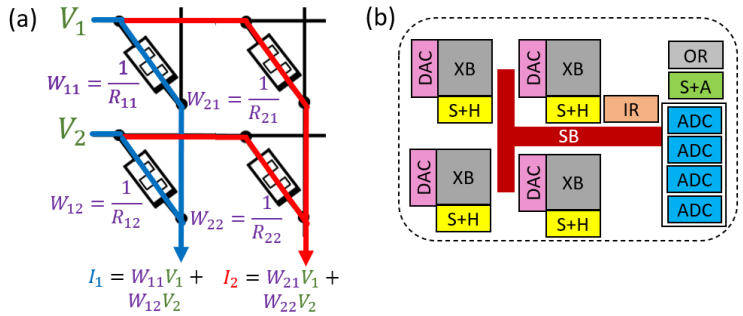


Fig. 1. Memristive analog computation and its use in the ISAAC architecture. (a) Analog VMM using memristive memory crossbar array. The input V_i is the voltage of row i and the output I_j is the current of column j. The values W_{ij} are stored in the memristors in the form of conductance. (b) ISAAC's IMA unit. IR - input register, OR - output register, S+A - shift and add, XB - memristive crossbar, S+H - sample and hold, DAC - digital to analog, ADC - analog to digital, SB - shared bus.

R_{ON} and R_{OFF}, and therefore can represent a bit value. This value is maintained when the power is shut down, rendering the memristor a NVM element. In a multi-level cell (MLC) mode [46], the memristor can have more than two possible resistances and as a result, can store multiple bit values.

Memristors are often arranged in a structure called a crossbar array [80], which is a memory array consisting of metal rows and columns. A memristor is connected to the row and column of each crosspoint in the crossbar. To avoid sneak path currents, a cell may also consist of an additional selector (either a nonlinear device or a transistor). Memristive crossbar memory arrays are typically extremely dense.

2.4 Analog Multiplication using Memristors

VMM can be efficiently performed using a memristive crossbar [79], as shown in Figure 1(a). The vector values are applied as voltages on the crossbar rows, and the matrix values are stored in the memristors in the form of conductance. The current in each memristor is determined (according to Ohm's law) by multiplication of the voltage on the memristor's row with its conductance. The current that flows in each column is the sum of all the currents in the memristors in this column (following Kirchhoff's Current Law, KCL). The performed calculation is therefore identical to a VMM and is executed concurrently in the entire crossbar array.

2.5 ISAAC's In-Situ Multiply Accumulate (IMA) Unit

ISAAC [64] is a convolutional NN accelerator based on memristive crossbars. Each ISAAC chip consists of tiles, which include several in-situ multiply accumulate (IMA) units. The IMA contains a few crossbar arrays of the same size (64×64, 128×128, or 256×256). Each crossbar cell stores a 2-bit value. To perform 16-bit multiplication, 16-bit value is distributed on 8 different cells and 16 calculation cycles are performed. In each cycle, 1-bit values are converted into voltages using digital to analog converters (DACs) that are applied on the crossbar rows (starting from the most significant bit [MSB] and ending with the least significant bit [LSB]). Then, the analog multiplication is performed using a crossbar array. The results are stored in sample-and-hold circuits, and converted using analog to digital (ADC) units. Results from all 16 cycles are shifted and summed up, using shift-and-add units. The IMA also contains registers and a shared bus, shown in Figure 1(b).

The main disadvantages of analog in-situ computation using a memristive crossbar are the high power consumption and area overhead caused by the ADCs. In Newton [53], improvements to the ISAAC IMA were proposed to reduce computations and ADC pressure. Some of the changes in the architecture are intra-tile optimizations, but some intra-IMA optimizations have also been proposed.

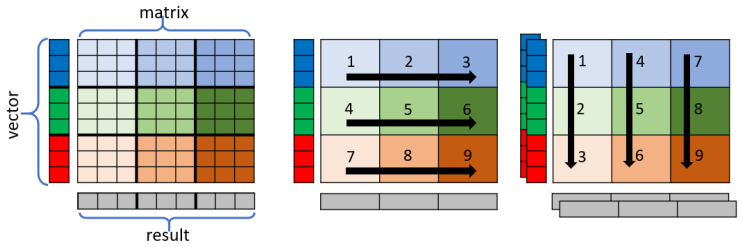


Fig. 2. Dataflows for maximizing data reuse in a memristive crossbar during multiplication.

Modifications of the HTree interconnect network were developed to reduce the system power. This is possible by placing constraints on the NN mapping to the tiles and IMAs. Adaptive ADCs, which tune the ADC resolution for power reduction without degrading the performance or the accuracy, were also proposed. Implementation of Karatsuba's divide-and-conquer multiplication technique was discussed to lower the power by reducing the use of ADC. Solutions to other challenges in RRAM computing systems, *e.g.*, accuracy and endurance, are also discussed in the literature [35, 50, 51]. We address the endurance issue in Section 6.5.

2.6 Software Solutions for VMM and MMM

Numerous software solutions for efficient VMM and MMM have been proposed. Level-2 BLAS [23] exposes vector-matrix operations to the compiler, to ensure efficient performance. Level-3 BLAS [22] was developed to support efficient matrix-matrix operations and other algebraic operations for architectures with a hierarchy of memory (*e.g.*, caches) or parallel processing architectures. The matrix is partitioned into blocks that can fit the cache, enabling data reuse without constant need to fetch it from the main memory. This technique was implemented and optimized in several tools, such as ATLAS [78] and BLIS [74].

3 EFFICIENT DATAFLOWS

The structure of the crossbar and the characteristics of memristors have given rise to the need for new dataflows and architectures. For example, the size of the matrices to be multiplied tend to be larger than the size of the crossbar. A typical size of a memristive array used as memory only, is 512x512 elements, whereas a memristive array used for analog computation, is usually smaller: 64×64, 128×128, or 256×256, due to the large and power-hungry periphery. Each memristor can store more than a single bit [46] (for example, two bits per cell in a 128×128 crossbar in the ISAAC IMA). The size of the multiplied matrix can reach thousands of fixed point numbers in one of its dimensions [32, 36]; therefore, one memristive crossbar cannot store the entire matrix. As a result, the matrix is divided into sub-matrices according to the dimensions of the memristive crossbar, as illustrated in Figure 2(a). Input vectors are divided into sub-vectors accordingly, and each sub-matrix is multiplied by the associated sub-vector. The matrix and vector tiling and order of execution are managed by the software.

While other multiplication dataflows also use tiling, our approach differs in several ways. First, some of the other dataflows focus on level-3 BLAS, since the potential for data reuse is significantly higher at this level ($O(n^3)$ operations over $O(n^2)$ elements). The crossbar can only perform VMM, and since MMM dataflows cannot be degenerated into sequential VMMs efficiently, they are not optimal in our architecture. On the other hand, existing VMM dataflows have prior assumptions on data patterns, *e.g.*, matrix sparsity, and use them for optimization. We target generic VMM dataflow and optimize data reuse. Second, most dataflows are designed for common architectures with cache

hierarchy and registers. Many parameters, *e.g.*, partition sizes and order of execution, are chosen to fit the specific architecture on which they operate. This causes the order of sub-matrix iterations to be different than those in our dataflow, which is intended for sequential VMM (memristive crossbar limitation) in the absence of cache hierarchy (PULP architecture limitation). For example, these dataflows always reuse sub-matrices, whereas in multiPULPly, this is not always optimal.

We consider two different scenarios: single-vector multiplication and multiple-vector multiplication. In each scenario, we set the order of multiplications differently to maximize data reuse and minimize data transfers. For a single vector, the matrix is scanned from left to right, row-wise, whereas for multiple vectors, the matrix is scanned from top to bottom, column-wise. The importance of selecting the dataflow most suitable to the scenario is demonstrated by the results. When multiplying a single vector with a matrix, the single-vector dataflow achieves 2.5× better speedup, as compared to the multiple-vector dataflow. In the multiple-vector scenario, the multiple-vector dataflow achieves the same 2.5× speedup, as compared to the single-vector dataflow. These dataflows are efficient thanks to the memristor non-volatility, which enables continued calculation without context restoring in case of a power failure.

3.1 Single Vector

In a real-time system, only a single sample is usually analyzed or classified at a time. For example, if the system's goal is to recognize a fingerprint, one sample arrives to the system and should be analyzed immediately. When only a single sample is analyzed by the NN, the FC layers perform one VMM. Therefore, in this case, efficient multiplication of a single vector with a matrix is required. In some NNs, *e.g.*, autoencoders, most of the layers are FC, and a specialized implementation of this specific scenario can accelerate the performance dramatically. Convolutional networks usually consist of a large number of CONV layers, which will be handled later on, and a small number of FC layers. Despite their small number, the FC layers consume a non-negligible part of the energy and runtime [13], and the dataflow can therefore accelerate such networks as well.

When multiplying one vector with a matrix, each sub-matrix can only be used once, whereas each sub-vector can be used more than once. Thus, data reuse is maximized by reusing the sub-vectors. In the single-vector case, the matrix is scanned row-wise so that all the sub-matrices corresponding to a specific sub-vector are executed one after the other, ensuring maximal sub-vector reuse. The pseudo-code is detailed in Algorithm 1. The variables *matNumOfRows* and *matNumOfCols* represent the number of matrix rows and columns, respectively. The variables *XBNumOfRows* and *XBNumOfCols* represent the number of crossbar rows and columns, respectively.

Algorithm 1 Single-Vector Dataflow Algorithm

```
N ← matNumOfRows / XBNumOfRows
M ← matNumOfCols / XBNumOfCols
// Iterate submatrices by rows
for i = 1 → N do
    subVec_i ← getSubVec(vec, i)
    configVec(subVec_i)
    // Iterate submatrices by columns
    for j = 1 → M do
        subMat_i_j ← getSubMat(mat, i, j)
        configMat(subMat_i_j)
        multiply(result)
        addResult(result, finalResult, i, j)
```

Figure 2(b) demonstrates the single-vector-matrix multiplication flow. The blue sub-vector is multiplied by several blue sub-matrices; the green sub-vector is multiplied by several green sub-matrices, and so on. First, the blue sub-vector is sent to the accelerator together with sub-matrix 1. Then, to multiply the blue sub-vector with sub-matrix 2, sub-matrix 2 is sent to the accelerator

and the blue sub-vector is not replaced. Then, sub-matrix 3 is transferred to the unit to perform a similar calculation. After the calculation is done, the green sub-vector replaces the blue sub-vector and is multiplied by the corresponding sub-matrices.

3.2 Multiple Vectors

In other common cases in real-time systems, several vectors must be efficiently multiplied with a matrix, *e.g.*, in CONV layers. Even when there is only a single input sample in the system, the CONV layers require multiple VMMs. Since the filter should be multiplied with different regions of the input matrix, the regions can be flattened and treated as separate input vectors.

In the multiple-vector scenario, each sub-matrix is multiplied by several sub-vectors. Both the sub-vectors and the sub-matrices can be reused; however, to maximize data reuse, we reuse the sub-matrices, which consist of additional elements. In this case, a sub-matrix is sent to the unit, and then its matching sub-vectors are sent, one after another; the matrix can be scanned either row-wise or column-wise. The chosen technique is iteration of the sub-matrices, column by column. By iterating the columns, intermediate results of different sub-matrices in the same column can be summed up in the accelerator, saving the energy that would be required to transfer these results back to the CPU, thus summing up the results in a parallel and efficient way. The concept is detailed in Algorithm 2.

Algorithm 2 Multiple-Vector Dataflow Algorithm

```
1:  M ← matNumOfCols / XBNumOfCols
2:  N ← matNumOfRows / XBNumOfRows
3:  // Iterate submatrix blocks by columns
4:  for j = 1 → M do
5:      // Iterate submatrix blocks by rows
6:      for i = 1 → N do
7:          subVecs_i ← getAllSubVecs(vecs, i)
8:          configVecs(subVecs_i)
9:          subMat_i_j ← getSubMat(mat, i, j)
10:         configMat(subMat_i_j)
11:         multiplyAllVecs(result)
12:         addResult(result, finalResult, i, j)
```

An example of the algorithm operation is demonstrated in Figure 2(c). The order of execution is as follows: sub-matrix 1 is sent to the unit with the blue sub-vectors; then sub-matrix 2 is sent with the green sub-vectors. Then, all the remaining sub-matrices are transferred to the unit in turn.

We present here only a simplified version of both algorithms. Other factors are considered and handled in the software code, *e.g.*, dividing the multiplication operations between the different crossbars, storing intermediate results in a buffer that resides in the accelerator, and complying with the TCDM size limitations by forming vector batches and operating on each batch separately. The register configuration is explained in Section 4.1.

Note that the data reuse in both algorithms is usually within a single sample. When several samples of the same neural network arrive at the network one after the other, we sometimes cannot utilize it for data reuse. In order to maintain the power constraints of our system, we use a size-limited TCDM, and we typically cannot store all the intermediate results in the accelerator or in the TCDM. This, of course, depends on the amount of samples and the size of the neural network.

3.3 The Tightly Coupled Data Memory (TCDM) Organization

The order of execution is managed by the software, which initiates VMM tasks and sends them to the accelerator. The TCDM stores the matrix and vectors of both engines, along with the results; therefore its address space is divided between them. Two TCDM configurations are implemented:

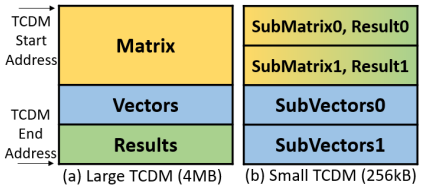


Fig. 3. TCDM organization.

large and small. In the large configuration, illustrated in Figure 3(a), the TCDM is sufficiently large to accommodate the matrix, vectors and results. On our set of benchmarks, 4 MB were adequate to store all this data. In this case, the hardware can directly access the data using the DMAs.

In the case of the small TCDM configuration (256kB), illustrated in Figure 3(b), since the matrix, vectors and results cannot fit into the TCDM, the multiplied sub-matrix and sub-vectors are constantly copied to the TCDM before the accelerator begins to work. The results overwrite the sub-matrices, since these values are no longer needed (as opposed to the large TCDM configuration, in which the matrix must be stored in the TCDM throughout the computation). If the sub-vectors do not all fit into the TCDM, additional iterations are performed until all the results are obtained.

4 HARDWARE ARCHITECTURE

The multiPULPly architecture is similar to the general HWPE structure, as illustrated in Figure 4. The accelerator includes a streamer, a control unit, and engines. All the white components in Figure 4 are HWPE original IPs that are used as is. The other parts were developed to fit the implemented computation task. In the control block, additional required registers were implemented, using the register file. In contrast to a regular HWPE, which has one FSM unit, one microcode processor and a single engine, the multiPULPly architecture includes two of each. The FSM unit and the microcode processor operate the engines. The number of engines was determined according to bandwidth limitations dictated by the architecture, as explained in Section 5.1. Each engine is responsible for performing the actual computation and was developed accordingly. The streamer, while containing only HWPE IPs, was built to support the required number of streams. A new block, a result adder, was added to the general architecture to manage the results coming from the engines and to save add operations in the processor, when possible.

4.1 Control

The control block manages the accelerator according to the control signals it receives from the peripheral port, which originate in the software. The general structure of the multiPULPly control block is similar to the general HWPE control unit and contains three sub-blocks: a register file, FSMs and microcode processors. However, some changes were made to these sub-blocks, as elaborated below.

4.1.1 Register File. In multiPULPly, we use several registers to control the execution flow. We use some of the HWPE general mandatory registers, which include an *acquire* register, to acquire the lock to offload a job, a *trigger* register, to trigger the execution of an offloaded job, and a *status* register, which returns the status of the HWPE (busy or idle).

Other user-defined registers were added to support multiPULPly functionality. These registers are used both to control the scenario flow, by changing between different states in the FSM, and to feed the engine with crucial data for the calculation. Each engine receives an enable signal coming from the register file. As explained in Section 4.2, only one engine is active in some cases; therefore the enable signals are not necessarily equal. We also pass on information about the matrix, sub-matrix, vector and sub-vector dimensions, addresses and strides to the DMAs and microcode

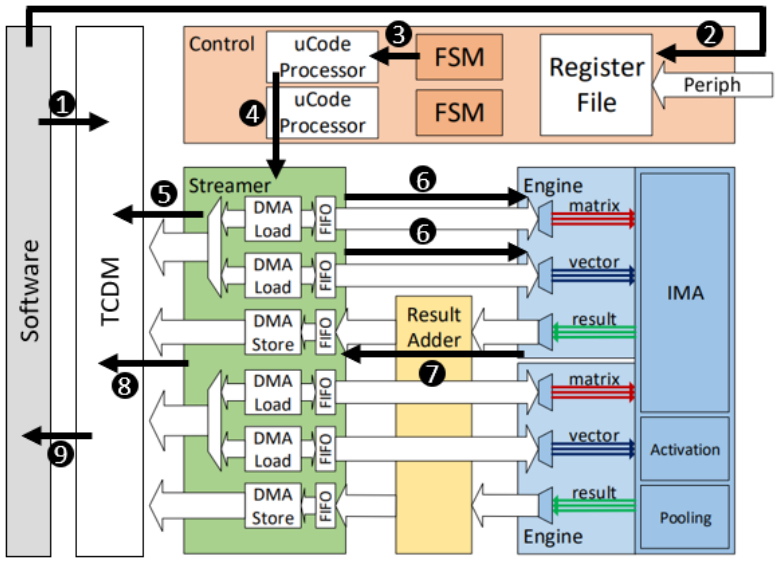


Fig. 4. multiPULPly hardware structure and execution flow.

controllers through the register file. Five additional control registers are also used and elaborated on in Table 2.

Table 2. Register Table

Register Name	Goal
Result buffer enable	The intermediate results of sub-matrices in the same column are stored/not stored in a buffer that resides in the engines during the multiple-vector scenario
Matrix transfer enable	A new matrix should/should not be pulled from the TCDM using the streamers
Result transfer enable	Transfer/do not transfer the result to the streamer and back to the TCDM
Add enable	Add/do not add the results from the different engines using the result adder
Pooling enable	Calculate/do not calculate a pooling layer

4.1.2 FSMs. The control block contains two identical FSMs, each operating a different engine. Since the FSMs are complex and contain 12 states and 24 transitions among them, we present here a simplified version[1] shown in Figure 5. In this section, we refer to the multiplied sub-matrix and sub-vector as "the matrix" and "the vector", respectively, for convenience.

The FSM operation consists of four main stages: matrix transfer, vector transfer, calculation and result transfer. The FSM enables procession through all or part of these stages according to the multiplication scenario. First, the FSM decides whether to start the matrix transfer stage. This stage can be skipped in the multiple-vector scenario, as described in Section 4.1. Next is the vector transfer stage, which can be skipped in the single-vector multiplication scenario. The calculation stage is always reached, in which the engine performs the actual VMM. When the calculation is complete, the FSM can either transfer the result or finish (in the case of result buffering). The vector transfer, calculation and result transfer stages are repeated, according to the number of vectors.

[1]The full FSM code is available in: https://github.com/adiha/multiPULPly.

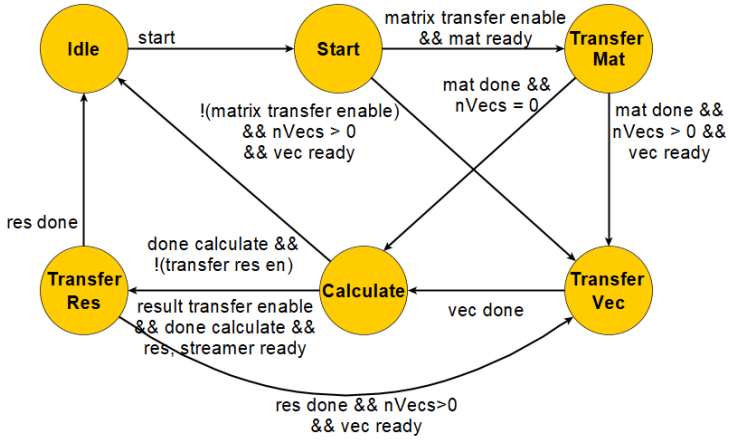


Fig. 5. The simplified FSM.

Matrix Microcode	Vector and Result Microcode

loop0:
-{
op: add,
a: mat_off,
b: mat_width,
}
	loop0:
	-{
	op: add,
	a: vec_off,
	b: mat_height,
	}
	-{
	op: add,
	a: res_off,
	b: mat_width,
	}

(a) The microcodes which control the streamers.

a[1]	a[2]	a[3]	a[4]
a[5]	a[6]	a[7]	a[8]
a[9]	a[10]	a[11]	a[12]
a[13]	a[14]	a[15]	a[16]

(b) An example of sub-matrix indices copied to the dot product unit by the microcode.

Fig. 6. The microcodes used in the FSM.

4.1.3 Microcode Processors. There are two microcode processors residing in the control block, one for each dot-product engine. They receive the heights and widths of the matrix and the sub-matrix, and the number of vectors from the register file. The microcode processor is responsible for outputting the offsets of the addresses to be copied from the memory to the engine, or vice versa, using the streamers. It receives the microcode from the FSM, according to its state. There are two possible microcodes implemented in the YAML format (Figure 6(a)), determined by the FSM: one for the sub-matrix and another for the sub-vector and result. They differ by the number of operations per loop and the number of iterations.

The matrix microcode has a single loop which adds the matrix width to the matrix offset each time, copying a matrix row at every iteration. As illustrated in Figure 6(b), to copy the green sub-matrix, the first row is copied using the sub-matrix offset. Then, the microcode adds the matrix width, which in this case is 4, to the offset, to copy the next row. The number of iterations of this loop is the sub-matrix height. The vector and result microcode has a similar loop responsible for advancing both the sub-vector and the result addresses.

4.2 Engines

The architecture includes two engines; the number of engines is determined by bandwidth limitations, as detailed in Section 4.4. The engines contain the actual datapath of the HWPE. Each engine receives a sub-vector and sub-matrix and multiplies them, using the ISAAC IMA unit [64]. The two engines stream their results to a result adder, which is responsible for adding the results of the different engines in the multiple-vector scenario.

The engine tracks the ready and valid signals of the input matrix and vector streamers, and when the data are ready for use, they are transferred to the IMA unit. Since in some edge cases the sub-matrix to be multiplied is smaller than the crossbar dimensions, the sub-matrix cannot simply be copied to sequential elements in the crossbar. This is similar to the case of copying a sub-matrix from a full matrix, as illustrated in Figure 6(b). Therefore, the indices are calculated to ensure that the sub-matrix is copied in its original form, which is important for the analog computation.

Each engine also streams the results from the IMA to the result streamer. It checks the ready and valid signals of the streamer to validate that it is ready to accept a new result, and confirms that the IMA has finished calculating. If both conditions are satisfied, the result is transferred to the streamer.

The engine also contains a result buffer of a configurable size. This buffer stores intermediate results and sends them back to the memory only when the calculation is complete. In the multiple-vector scenario, the multiplication results of sub-matrices in the same column are eventually supposed to be summed up. Performing the summation in the hardware is faster and allows the accumulative result to be saved in this buffer and sent back to the memory when the last matrix in the column has finished calculating. We do not assume that the buffer is large enough to store all the multiplication results for all the vectors. Hence, the results that fit into the buffer are saved in it and returned at the end of computation; all other results are immediately sent to the memory via the streamers.

To enable full neural network execution, the engine includes a pooling unit and an activation function unit. The pooling unit supports max pooling. Average pooling is also supported by normalizing the weights. The activation unit supports the ReLU function.

4.3 Result Adder

In the multiple vector scenario, the result adder is responsible for adding the results coming from the engines, since the multiplication results of sub-matrices in the same column are supposed to be summed. The adder block has two stream inputs, each belonging to one of the engines. The block outputs are the original or summed results, depending on the scenario. In the multiple-vector case (Figure 2(c)), where the sub-matrices are iterated column by column, the results of different sub-matrices can be summed inside the unit, whereas in the single-vector case, where the sub-matrices are iterated row by row (Figure 2(b)), the results are not summed. The block reads the ready and valid signals that come from the input streams, which implement a simple AXI stream-like protocol. According to the input signals, the block deduces when the results are ready to be read. If only one dot product engine is active (*i.e.*, there is an odd number of sub-matrices and this is the last iteration), the result is transferred to one of the output streamers, and the ready and valid signals are set accordingly. The ready and valid signals of the other streamer are also set accordingly, and therefore, no data are sent. If both engines are active, the adder checks whether the results need to be added according to the add enable register. This is determined by the scenario in the software, *i.e.*, when both engines process sub-matrices in the same column. In this case, results from both streams are added and transferred to one of the output streamers. In other cases, the results are transferred separately to an appropriate output streamer.

4.4 Streamer

As shown in Figure 4, the streamer marked in light green includes six DMAs: a matrix DMA, a vector DMA, and a result DMA for each dot-product engine. The DMAs transfer the data from the TCDM to the accelerator in a balanced manner. The matrix and vector DMAs transfer data from the TCDM to the engine using a load unit, and the result DMAs transfer data from the engine to the TCDM using a store unit. The DMAs receive addresses of data bursts from the microcode processor

in the control block, and the burst sizes from the control FSM. The streamers do not perform data re-layout, since there are no assumptions on data characteristics. In [20], a sparse matrix reordering technique for memristor-based multiplication is proposed. It achieves 90% performance improvements. Using the proposed technique in our system would yield a 1.1× speedup, according to Amdahl's law.

4.5 Application Offloading

The application offloading is performed in several steps, as shown in Figure 4. First, if necessary (*i.e.*, in small TCDM configuration), the data is written by software to the TCDM (①). Then, our API functions for single and multiple vector multiplication set the registers, according to the multiplication scenario (②). In phases ③ and ④, the FSM loads the different microcodes onto the processors, which output the address offsets to the streamer. Consequently, the streamer pulls the data from the TCDM (⑤) and passes it on to the engines (⑥). When the calculation is done, the results are summed (if necessary) in stage ⑦ and are then written back to the TCDM (⑧), and can be read by software (⑨).

5 METHODOLOGY

5.1 Architecture Choice

Table 3 lists the different components in the architecture. We describe here these design choices and their motivations. The technology and frequencies were determined according to [61]. Since we target IoT applications, a single core was used. A large number of engines/crossbars can reduce the latency and the energy of the calculation, especially when there is a relatively large number of vectors. However, due to bandwidth limitations, the HWPE design enables only four ports from the streamer unit to the TCDM. Since every engine requires two ports, an input port (for matrix and vector values) and an output port (for result values), a total of two engines can be used in the accelerator, each using two ports. In addition, most of the accelerator power was contributed by the IMA, as discussed in Section 6.4. The addition of extra crossbars increased the power consumption dramatically. Therefore, we limited the size of the crossbars and their number to one or two. Due to clock gating, when the accelerator is active, the other parts of the core are not, allowing the system to maintain a power envelope of a few milliwatts. We set the TCDM size to 4 MB, which is the smallest size that enables use of the large TCDM configuration demonstrated in Figure 3, with our set of benchmarks. The importance of using this configuration is demonstrated in Section 6.

Table 3. Architecture Component Specification

Component	Number/Size/Value
Technology	22nm CMOS
Frequency Range	150MHz-670MHz
Core	Single 32-bit in-order RISC-V Core
Number of Crossbars/Engines	1 or 2
Crossbar Size	64x64, 128×128 or 256×256
Number of DMAs	6
TCDM Size	4 MB

5.2 Modeling ISAAC IMA

To emulate the functionality and delay of the ISAAC IMA unit [64] in the system, we replaced it with our design in both simulations and on the FPGA. For the delay calculation, we developed a simple block which imitates the ISAAC IMA delay, but does not supply its functionality. For the functionality check, we developed a dot product unit, shown in Figure 7, which represents the full IMA. The dot product unit has three main components:

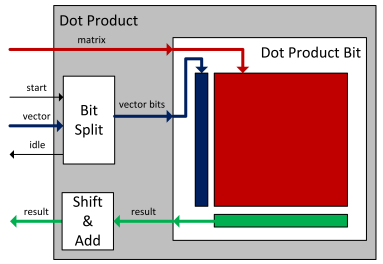


Fig. 7. The dot product unit which replaces the ISAAC IMA in simulations.

(1) **Dot Product Bit**: This block gets a vector of bits (one bit from each vector element) and a matrix, and multiplies them. It replaces the memristive crossbar functionality. For performance evaluation, the block is implemented so that it follows the ISAAC IMA unit latency.
(2) **Bit Split**: This unit splits the input 16-bit vector into 16 vectors of 1-bit elements, one clock cycle each. As mentioned in [64], when applying 16-bit voltage values on each row of the crossbar, large ADCs are needed. Therefore, we apply a 1-bit voltage value in each clock cycle to reduce the ADC area and power. The unit output is used as an input to the dot product bit block.
(3) **Shift and Add**: This block is responsible for adding the results of different voltage bits. In the first cycle, the MSB vector voltage bits are fed into the dot product bit unit by the bit split. When the result is ready, it arrives at the shift and add unit, which stores it. In the next cycle, the next bits are fed into the dot product bit unit, producing new results. Then, the shift and add unit shifts the previous result to the left and adds the new results, and so on.

5.3 Measurements

For performance evaluation of the processor and the accelerator (not including the IMA), we use the Xilinx Zynq-7000 SoC ZC706 Evaluation Kit with HERO [44], an FPGA-based platform that consists of programmable accelerators with RISC-V core clusters and an ARM processor. HERO allows us to check the performance reliably and quickly (faster than simulation), using performance counters. The system operates at a 50MHz frequency, and includes one cluster with a single core, a 4 MB TCDM and a 4 kB cache.

For area and power estimation of the PULPissimo processor, we used numbers from [61]. For the HWPE estimation, we used the Synopsys Design Compiler with the 65nm LP LowK technology scaled to 22nm, according to the polynomial model suggested in [67, 68]. For the ISAAC IMA design part, we used numbers from [64] and [43]. We assumed that, due to the clock gating mechanism in PULP, both PULP and the IMA consume power only when they are active.

5.4 Benchmarks

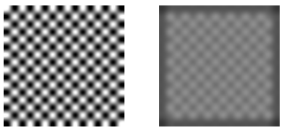
First, we selected layers from different mobile NNs as benchmarks, to show the acceleration within a layer, depending on its type and the used multiplication algorithm. The layers are listed in Table 4, where m and n are the matrix height and width and p is the number of vectors. The matrix and vector sizes varied from hundreds to thousands of elements (to assess different scenarios; some matrices fit into the crossbar, whereas some did not). The layer types, FC or CONV, represent the single and multiple vector cases, respectively. Later on we also show a full neural network execution of mobileNet [32].

6 RESULTS

In this section, we evaluated the system in terms of functionality, performance, energy and area, and compared it to state-of-the-art. We separated the architecture into two parts: the ISAAC IMA

Table 4. Neural Network Layer Benchmarks

Network Name	Layer Type	m	n	p
MobileNet [32]	FC	1024	1000	1
EdgeSpeechNet [49]	FC	45	12	1
NU-LiteNet [73]	FC	196	50	1
SqueezeNet [36]	FC	196	1000	1
MobileNet [32]	CONV	27	32	12544
MobileNet [32]	CONV	256	256	784
MobileNet [32]	CONV	512	1024	49
NU-LiteNet [73]	CONV	9	64	3136



(a) Original image (b) Filtered image

Fig. 8. Functionality check: filtering using a mean filter.

and the remainder of the logic. These two different parts were evaluated separately in each of the metrics.

6.1 Functionality Check

We developed several tests, which include edge cases, to check whether the multiplication results are correct. We compared the results with a Matlab model and verified that the results are bit-accurate. An example of a convolution of an image with a mean filter is displayed in Figure 8.

6.2 Performance Evaluation

We ran the benchmarks with four different solutions:

(1) **Baseline**: Naive MMM or VMM, which runs on PULP. In the algorithm below, A ($n \times m$ matrix) and B ($m \times p$ matrix) are the input matrices and C is the output matrix.

Algorithm 3 Naive Matrix-Matrix Multiplication

```
1: for i = 1 → n do
2:     for j = 1 → p do
3:         C_ij ← 0
4:         for k = 1 → m do
5:             C_ij ← C_ij + A_ik · B_kj
```

(2) **BLAS**: Using BLAS [22, 23] for efficient MMM or VMM. Since PULP supports only pure C programs, and the implementation should be compatible with the RISC-V ISA, we used a simple RISC-V BLAS implementation. Other implementations, which adapt the algorithm to the endpoint characteristics [74, 78], such as the cache size, might yield better performance; however, we predicted that, in practice, this improvement will not be significant in our architecture.

(3) **Single**: Using multiPULPly's single-vector function.

(4) **Multiple**: Using multiPULPly's multiple-vector function.

We performed a full design exploration by running the benchmarks with the different dataflows, while changing the number and size of the memristive crossbars. Our analysis considers the write operations to the memristive crossbar arrays. A read operation is one order of magnitude faster than a write operation in the 1T1R cell [81]. Assume a row-by-row crossbar array programming, in a layer where there are numerous vectors to multiply, the write latency becomes negligible compared to the overall latency (up to 8% of the overall latency for 1000 vectors or more). In

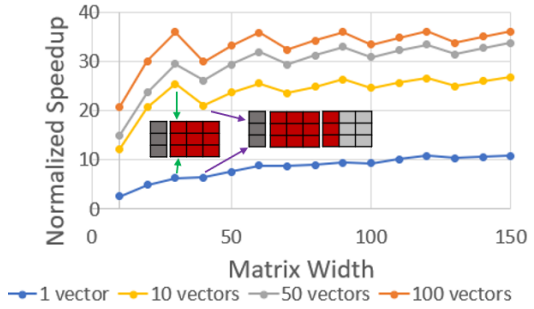


Fig. 9. Normalized speedup of VMM and MMM with respect to naive implementation for different matrix widths.

addition, to reduce the write overhead, the programming operates in parallel to the weight transfer from the TCDM to the accelerator, which is performed gradually. From energy perspective, the write operation is roughly three times more energy consuming than the read operation [81]. We count the number of writes and add it to the consumed energy. Similarly, as the number of vectors grows, this energy is negligible compared to the overall computation. We also determined how the TCDM size affected the performance.

First, we evaluated the influence of the matrix size on the speedup with synthetic examples. Figure 9 shows the normalized speedup of VMM and MMM for a matrix with a fixed number of rows (128) for a different number of columns using a single crossbar array. The results show that, in general, as the size of the matrix grew, the speedup increased, since memristive crossbar parallelism was utilized. The local peaks in the graph (marked with green arrows) were achieved when the crossbar was fully utilized (the matrix is the exact size of the crossbar or multiple crossbars), and a small decrease in speedup was identified when the crossbar cells were poorly utilized (marked with purple arrows).

Next, we evaluated the influence of the TCDM size on the performance. The results of a 256kB-sized TCDM and a non-limited sized TCDM with two 128×128 crossbars are shown in Figure 10. The multiPULPly dataflows are marked as "multiple" for the multiple vector dataflow and "single" for the single vector dataflow. The geometric mean was calculated according to the relevant benchmarks (FC benchmarks for the single-vector dataflow, and CONV benchmarks for the multiple-vector dataflow). The non-limited sized TCDM configuration achieves 18× better speedup, as compared to the 256kB-sized TCDM configuration, meaning that the overhead of the extra copy cycles is non-negligible. In our simulations of the small TCDM configuration, using the HERO environment, we update the content of the TCDM from the ARM processor. The performance could be improved by directly connecting an off-chip memory to the TCDM bus (using DMAs). However, even the most efficient data transfer from an off-chip memory would still introduce a non-negligible performance and energy overhead compared to the large TCDM configuration, and would not comply with the IoT demands and limitations. Therefore, for the rest of the analysis, we use a sufficiently large TCDM to accommodate all the needed data.

Figure 11 shows the normalized speedup for the different benchmarks in Table 4. The multiPULPly dataflows are marked as "multiple" for the multiple vector dataflow and "single" for the single vector dataflow, together with the crossbar size: 64×64, 128×128 and 256×256. Each benchmark was run with configurations of one and two crossbars (bottom bars with lighter colors represent the one-crossbar configuration, and top bars with darker colors represent the two-crossbar configuration). As expected, for the FC layers, the single-vector dataflow achieved better results (2.5×); and, for

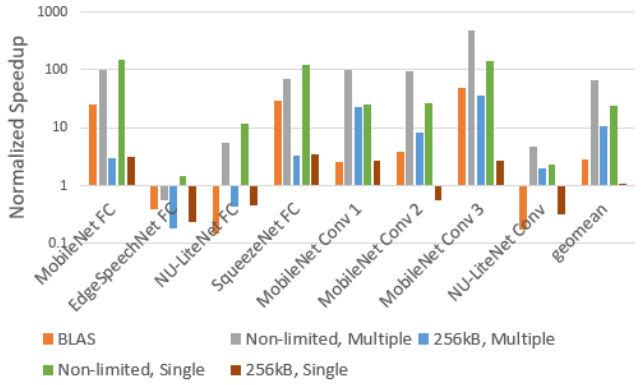


Fig. 10. Normalized speedup with respect to a naive implementation for different TCDM sizes with 128×128 multiple crossbars at 150MHz.

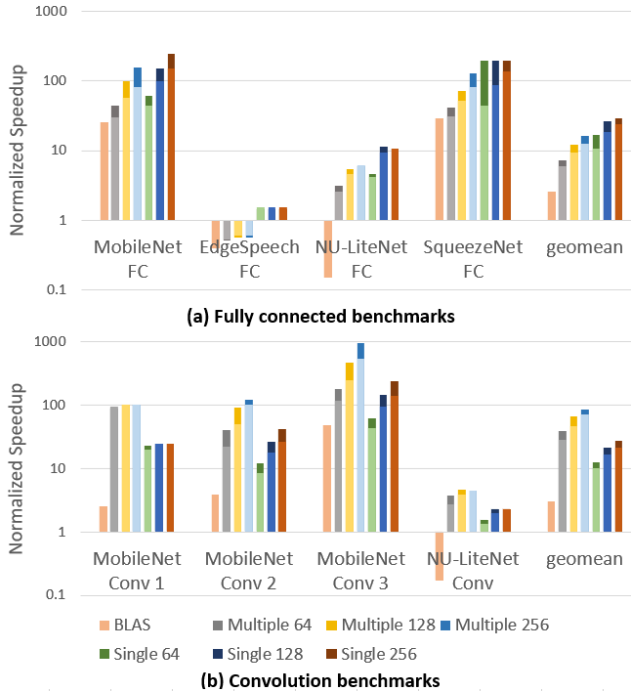


Fig. 11. Normalized speedup of different layers (FC on upper graph and CONV on lower graph) with respect to the naive implementation for different numbers (lighter - 1 crossbar, darker - 2 crossbars) and sizes of crossbars at 150MHz.

the CONV layers, the multiple-vector dataflow showed better performance. In general, as the crossbar size grew, the performance improved. When the multiplied matrix was small, enlarging the crossbar did not improve the performance. The two-crossbar configuration achieved better results in comparison to the single-crossbar configuration, when the matrix was large enough. In addition, use of BLAS, the software solution, on PULP, degraded the performance for small benchmarks, since the overhead of using BLAS is significant for small matrices. In contrast, multiPULPly always achieved better performance when using the appropriate dataflow. Our accelerator achieved 51× better speedup than the naive software implementation, and 27× better speedup than BLAS.

Note that our architecture targets inference of a specific neural network at a time as usually required in IoT applications, and therefore, every benchmark was run on our architecture separately. In our analysis, we assume that the weights are initially stored in the TCDM, as it functions as the main memory in our system. Because of the power limitations, typically, the TCDM is not large enough to accommodate more than one network weights. Switching applications would require replacing the TCDM content. which is possible in other PULP architectures that include off-chip memory [44], but not in our IoT-targeted architecture.

6.3 Area Evaluation

The area of each component is detailed in Table 5 for two 128×128 crossbars in 22nm CMOS technology. The IMA area is negligible in comparison to the total area of the accelerator and the total area of the system. Because of bandwidth and power limitations, we added one IMA with only two memristive crossbars to the system. Overall, the total area overhead was 4% (the IMA area overhead was 0.1%).

Table 5. Architecture Power and Area Analysis.

Component	Param	Spec	Power (uW)	Area (um^2)
IMA [64] [43]				
ADC	resolution frequency number	8 bits 1.2GSps 2	600	1000
DAC	resolution number	1 bit 2 x 128	100	19
S+H	number	2 x 128	0.3	4
Crossbar	number size bits per cell	2 128 x 128 2	80	20
S+A IR OR	number size size	1 0.5 KB 64 B	6 40 8	27 238 87
IMA Total			834.3	1395
Other				
HWPE Logic			34.3	56,000
PULPissimo			1350	1,220,000

6.4 Energy Evaluation

The power consumption for each component is detailed in Table 5. The processor components are divided into two categories: *IMA* and *Other*. The *IMA* part consists of the IMA with the crossbar arrays, specifically, two 128×128 crossbars in a 22nm CMOS technology at 25°C, with a 150MHz operating frequency. The *Other* part includes the accelerator logic (HWPE logic), including all its logic components (result adders, streamers, *etc.*), and the processor itself (PULPissimo). In addition, the write operations to the memristive crossbar arrays for the weight updates are considered [82].

The most power-consuming component in the IMA, which limits the overall energy efficiency, is the ADC [64]. This is a drawback of our (and every memristive analog-based computing) architecture. The difference between our architecture and other designs which use memristive analog computation, is that in our architecture, we limit the amount of the ADCs and the periods of time in which they are active by the gating technique. In other architectures, *e.g.*, ISAAC, these components have to be active at all times, because of the pipelining applied in it. In this manner, we significantly reduce the power in our system, which is a critical factor in IoT systems, and also improve the energy efficiency. Still, the ADC energy is 50% of the overall consumed energy in ISAAC, and in our architecture it is 40%. A lower sampling rate than used in our system can be also applied. This would degrade the performance.

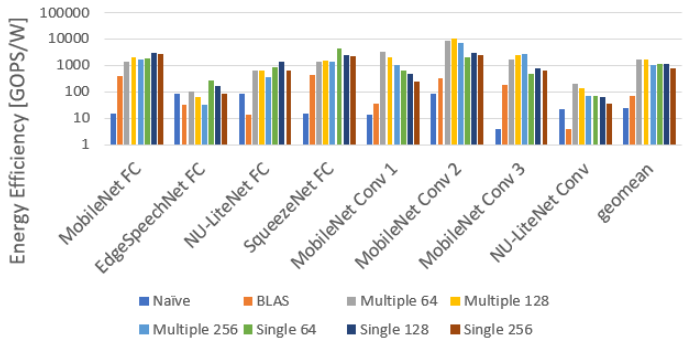


Fig. 12. Energy efficiency for different multiPULPly configurations with multiple crossbars at 150 MHz.

One technique that was suggested in ISAAC and also used in our architecture, is encoding to reduce the ADC size. Every w-bit synaptic weight in a column is stored in its original form, or in its "flipped" form. The flipped form of w-bit weight W is represented as $\bar{W} = 2^w - 1 - W$, and the analog computation is performed with $\bar{W}$. This requires storing an additional bit per column, but the ADC size requirement is lowered by one bit.

The modifications to the ISAAC IMA, suggested in Newton [53], could not be applied in this study for several reasons. HTree modifications using weight mapping constraints and Karatsuba's divide-and-conquer algorithm are irrelevant, since we use a small number of crossbars. Adaptive ADCs, also suggested in Newton, can be used in our architecture. However, the power of the adaptive ADCs is not detailed; therefore, we could not easily adopt this modification and it is left for future work.

Figure 12 shows the energy efficiency for different benchmarks. Using the most suitable dataflow, *i.e.*, multiple-vector for CONV layers and single-vector for FC layers, results in better energy efficiency than all the other solutions. Enlarging the crossbar dimensions can increase the energy efficiency for large-sized matrix benchmarks, but decrease the energy efficiency for the small-sized matrix benchmarks (since the large crossbar consumes more energy, but does not contribute additional parallelism). The dataflow can be extended to handle small matrices by duplicating them in the crossbar. In this manner, parallelism is improved, and consequently, so are the energy efficiency and performance. Our accelerator achieved 55× better energy efficiency than the naive software implementation, and 19× better energy efficiency than BLAS.

6.5 Endurance Evaluation

The analog multiplication was performed by reading the memristors, and the write operations only occurred when the weights were replaced. For the largest CONV benchmark and medium-sized crossbar, we wrote to each cell 16 times in 5 milliseconds. The write intensity was 10× higher for the largest FC layer as comapared to the CONV layer. However, the unit is not active and performing FC/CONV operation nonstop [100% duty cycle]. For 10% duty cycle, the hardware will be valid for approximately 10 years [33]).

6.6 Comparison to State-of-the-Art

We compared multiPULPly to state-of-the-art low-power CONV accelerators operating at different voltages. We did not compare multiPULPly to RRAM accelerators, since they use many crossbars, and therefore do not target low power (*e.g.*, ISAAC [64]: 65.8W, PUMA [8]: 62.5W). They cannot be degenerated to use only several crossbars since they rely on storing all the weights.

We hereby bring a high-level estimate of ISAAC with clock gating mechanism and comparison of such architecture to multiPULPly. One issue with gating in ISAAC is that ISAAC uses pipelining, *i.e.*, all the layers are executed in parallel, and therefore, it activates the memristive crossbar arrays that participate in the execution at all times. This means that gating would be only beneficial for deactivating arrays which are not needed for the network at all. For the more compact mobile neural networks, some crossbar memristive arrays can be deactivated. Since we target mobile neural networks, which are significantly smaller, the benchmarks tested on multiPULPly were not tested in ISAAC original work, and we can only perform high-level estimate of the energy efficiency of ISAAC for mobile neural networks.

The ratio between the number of crossbar arrays in ISAAC and the actual number of needed arrays in a mobile neural network depends on the ISAAC configuration (as this is a hierarchical architecture that can contain different number of tiles, IMAs and crossbar arrays) and the size of the mobile network. We compare here the ISAAC configuration that yields the best energy efficiency with the basic mobileNet configuration, which includes 4.2 million parameters. In this case, the accommodated crossbar arrays will be reduced by 20×. The power will be reduced by a lower factor than 20×, since there are components which are shared by multiple arrays, and because of the gating mechanism. The number of calculations will roughly be reduced by 20×. Therefore, even if we assume that the power will be reduced by 20×, the energy efficiency stays the same (644 GOPS/W), which is inferior to that of multiPULPly (19.5 TOPS/W). Even after scaling to the same technology, the energy efficiency would not be better than this of multiPULPly.

In terms of performance, the throughput is not affected by the clock gating (41.3 TOPS for the ISAAC-CE configuration [64] - best computational efficiency). multiPULPly (82 GOPS) cannot compete ISAAC, since ISAAC performs all the computations in parallel. In terms of area, the ISAAC chip is 70× larger than multiPULPly, without considering the clock gating overhead.

For multiPULPly, we used the PULP operating points in [61]. Table 6 compares the energy efficiency and peak performance of multiPULPly and other works, both in their original design and scaled to 22nm CMOS technology. Curly brackets indicate numbers scaled to 22nm CMOS technology. ShiDianNao was not scaled, because the voltage was not mentioned in the paper. *LV* and *HV* indicate a low or high voltage configuration, respectively. The peak performance was determined as explained in [18], and was achieved for a 256x256 crossbar configuration, when the matrix size was exactly the size of two crossbars. In terms of peak performance, multiPULPly outperformed HWCE [18]. However, it exhibited lower peak performance than Origami [14], since the crossbar was relatively slow compared to the operating frequencies. The multiPULPly energy efficiency was superior to the state-of-the-art. Even after scaling, multiPULPly displayed 1.5× better energy efficiency than Origami, and 4.5× better energy efficiency than Chain-NN. Other low-power accelerators achieved better energy efficiency [6, 52], but supported only BNNs. Moreover, multiPULPly accelerates FC layer computation, in addition to the CONV layer.

Table 6. Comparison to State-of-the-Art Low-Power and Ultra-Low-Power Accelerators

Accelerator	Tech.	Volt. [V]	Freq [MHz]	Power [mW]	Peak Perf. [GOP/S]	Peak Energy Eff. {22nm Scaled} [TOPS/W]
ShiDianNao [25]	65nm	-	1000	320	194	0.6 {-}
Eyeriss [15]	65nm	1	200	287	46	0.16 {2.3}
ChainNN [76]	28nm	0.81	700	567	806	1.42 {4.31}
HWCE LV [18]	28nm	0.4	22	0.7	1	1.37 {7.75}
HWCE HV [18]	28nm	0.8	400	142	37	0.26 {0.77}
Origami LV [14]	65nm	0.8	189	92	74	0.8 {12.51}
Origami HV [14]	65nm	1.2	500	445	196	0.44 {6}
multiPULPly LV	22nm	0.5	150	1.5	30	19.5 {19.5}
multiPULPly HV	22nm	0.8	670	8	82	10.3 {10.3}

We also executed a full neural network, mobileNet [32], on multiPULPly, and compared it to execution of the network on state-of-the-art commercial microcontroller units (MCUs). The input image was of size 160 × 160 and width multiplier $\alpha = 0.25$. The depthwise separable layers are an integral part of MobileNet, and include two sub-layers: 3×3 depthwise convolution, in which every channel is convoluted separately, and 1×1 convolution, which merges all the channel outputs into one output. In the depthwise convolution sub-layer, the kernels are smaller and therefore the sum-up granularity is smaller. Unlike other architectures which store the kernels in the crossbar arrays and do not replace them throughout the computation, we have the freedom to choose to use either the layer input or the kernel as the matrix values, and use the other as the vector or vectors. In this case, we map the kernel as the vector and the input as the matrix values to achieve better parallelism. Since the vector is typically shorter than the crossbar array height, not all the matrix rows are occupied, so we pad the vector with zeros. In this way, we do not override older matrix values that are not used in the multiplication and save costly write operations. The matrix columns, however, are all occupied, as the kernel is multiplied by different parts of the input channel and massive parallelism is achieved. It is also important to mention that these sub-layers constitute only 3% of the MAC operations in MobileNet.

The second sub-layer, 1×1 convolution, is a standard convolution that constitutes 95% of the MAC operations. The number of channels of the input in these layers grows as the network becomes deeper. Therefore, the kernel becomes longer in one dimension, better utilizes the crossbar array height, and enlarges the sum-up granularity. In addition, since several kernels are multiplied by the same input channel, as many kernels as possible are mapped to the crossbar array in different columns, to further utilize its area and computation capabilities.

The results appear in Table 7 for the different PULP operating points and for the naive, BLAS and multiPULPly implementations. The number of cycles is different between the operating points since the memristive crossbar operates in a different frequency. The frames per second (FPS) value is higher using GAP9 architecture compared to multiPULPly as it uses a larger number of cores, and its active power is higher, accordingly. However, the wasted energy is 4 × lower in multiPULPly. The consumed energy values are larger using PULP without multiPULPly by one or two orders of magnitude; therefore, this results prove our accelerator effectiveness.

Table 7. Full Network Comparison of multiPULPly to State-of-the-Art Commercial Microcontrollers

Target	Freq [MHz]	Time [ms]	Cycles [M]	FPS [frames/sec]	Active Power [mW]	Consumed Energy [mJ]
STM32 H7 [4]	400	162.5	65	6.2	170	27.6
GAP9 [1]	29	162.5	4.77	6.2	5	0.8
GAP9 [1]	400	11.925	4.77	83.9	50	0.6
PULP [60] Naive Implementation	150	6960.4	1044	0.14	1.5	10.4
PULP [60] BLAS Implementation	150	1575.8	236	0.63	1.5	2.4
multiPULPly LV	150	66.8	10.02	14.97	1.5	0.1
multiPULPly HV	670	14.74	9.88	67.84	8	0.1

As discussed in Section 3.2, we run each sample on a neural network at a time since the TCDM is not always sufficiently large to accommodate the intermediate results of all the samples. Figure 13 shows the speedup of running a batch of samples in the case of mobileNet, taking advantage of the data reuse that can be achieved (with inter-sample data reuse), with respect to the sample-by-sample computation (without inter-sample data reuse). The TCDM can be enlarged at the cost of increasing the architecture power to support such speedup.

7 RELATED WORK

Since ISAAC [64], much work has been devoted to accelerating NNs, using memristive crossbars [40, 47, 53, 72]. However, these systems are not suitable for IoT, since the large number of memristive

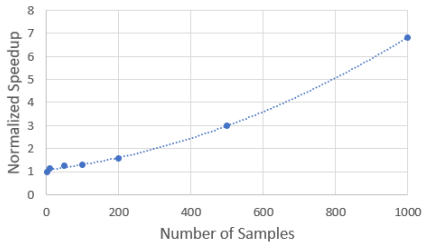


Fig. 13. Normalized speedup of sample batch computation with respect to sample-by-sample computation in MobileNet.

crossbars, accommodating all weights, result in high energy consumption. The consumed power in ISAAC [64] is 65W, whereas multiPULPly does not exceed a power envelope of a few milliwatts.

Attempts have been made to design low-power NN accelerators [58], some of which use memristors [27]. However, only a few rely on data reuse to minimize the consumed energy. In Eyeriss [15], a dataflow is proposed to minimize the energy consumption incurred by data spatial architectures by reusing filter weights. Chain-NN [76] reduces the bandwidth and the energy consumption by reusing input operands. However, both use volatile memories; as such, they are not suitable for ultra-low-power architectures such as PULP, which uses a normally-off policy to tolerate frequent power failures. Leveraging sparsification on DNNs was widely investigated in the literature [54, 75, 83]. In [10], the sparsification of FC layers is leveraged to reduce the resource requirements on wearables. However, their solutions are data-type-dependent and reduce the accuracy.

All the aforementioned low-power accelerators significantly exceed the power limitations of IoT devices. HWCE [18], an accelerator for PULP, was designed to efficiently perform convolution. Origami [14] is an accelerator that achieves a high energy efficiency of 1.421 TOPS/W. These accelerators, however, scale poorly to large kernels and inputs. In addition, they accelerate fixed-sized CONV layers, do not accelerate FC layers, and do not use NVM. In [56], a System-on-Chip is proposed for convolution network acceleration. It accelerates both CONV and FC layers, but achieves similar energy efficiency to Origami. A 0.62mW NN face recognition processor was suggested in [11]. It has lower energy efficiency than multiPULPly and is application-specific, while we designed an accelerator for a general-purpose system.

In [9], a mapping method of the Binary Level-1 BLAS on a memristive crossbar array using stateful logic is suggested. They address binary vector operations only, whereas our NN-dedicated solution efficiently performs analog VMM.

Another solution is the software programmable architectures, which provide more flexibility in neural network execution. Some of these MCUs fit the power envelope of IoT devices, *e.g.*, ARM's STM32H7 family [4] and Greenwaves' GAP9 [1]. Many optimized software libraries have been developed for these architectures to cope with the limited computational and memory capabilities, *e.g.*, CMSIS-NN [45] for the STM32H7 family. Similar solutions have been proposed for GAP9.

8 CONCLUSIONS

This paper presented multiPULPly, a neural network accelerator for PULP, which uses a memristive crossbar for efficient multiplications. The benefits of the crossbar-based accelerator are twofold: compatibility with the normally-off policy, and parallel computation. We also showed novel dataflows to reduce energy consumption. The proposed accelerator achieved better energy efficiency, compared to state-of-the-art accelerators, and enhanced the performance of neural network applications, as compared to existing accelerators and software solutions. While not shown in this paper, multiPULPly can accelerate other applications which perform vector-matrix multiplication, *e.g.*, graph analytics, assuming limited accuracy is acceptable.

REFERENCES

[1] [n.d.]. GAP9. https://greenwaves-technologies.com/gap9_iot_application_processor.

[2] [n.d.]. Pulp Platform Website. https://www.pulp-platform.org.

[3] [n.d.]. YAML. https://yaml.org.

[4] 2018. *stm32h743 datasheet.*

[5] H. Akinaga and H. Shima. 2010. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proc. IEEE* 98, 12 (Dec 2010), 2237–2251. https://doi.org/10.1109/JPROC.2010.2070830

[6] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. 2018. YodaNN: An Architecture for Ultralow Power Binary-Weight CNN Acceleration. *TCAD* 37, 1 (Jan 2018), 48–60. https://doi.org/10.1109/TCAD.2017.2682138

[7] Miguel Angel Lastras-Montaño and Kwang-Ting Cheng. 2018. Resistive random-access memory based on ratioed memristors. *Nature Electronics* 1 (08 2018), 466–472. https://doi.org/10.1038/s41928-018-0115-z

[8] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-Mei Hwu, John Paul Strachan, Kaushik Roy, and Dejan S. Milojicic. 2019. PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference. *CoRR* abs/1901.10351 (2019). arXiv:1901.10351 http://arxiv.org/abs/1901.10351

[9] D. Bhattacharjee, F. Merchant, and A. Chattopadhyay. 2016. Enabling in-memory computation of binary BLAS using ReRAM crossbar arrays. In *2016 VLSI-SoC*. 1–6. https://doi.org/10.1109/VLSI-SoC.2016.7753568

[10] Sourav Bhattacharya and Nicholas D. Lane. 2016. Sparsification and Separation of Deep Learning Layers for Constrained Resource Inference on Wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM* (Stanford, CA, USA) *(SenSys '16)*. ACM, New York, NY, USA, 176–189. https://doi.org/10.1145/2994551.2994564

[11] K. Bong, S. Choi, C. Kim, S. Kang, Y. Kim, and H. Yoo. 2017. 14.6 A 0.62mW ultra-low-power convolutional-neural-network face-recognition processor and a CIS integrated with always-on haar-like face detector. In *2017 ISSCC*. 248–249. https://doi.org/10.1109/ISSCC.2017.7870354

[12] Robert A. Bridges, Neena Imam, and Tiffany M. Mintz. 2016. Understanding GPU Power: A Survey of Profiling, Modeling, and Simulation Methods. *ACM Comput. Surv.* 49, 3, Article 41 (Sept. 2016), 27 pages. https://doi.org/10.1145/2962131

[13] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. 2017. NeuralPower: Predict and Deploy Energy-Efficient Convolutional Neural Networks. *CoRR* abs/1710.05420 (2017). arXiv:1710.05420 http://arxiv.org/abs/1710.05420

[14] L. Cavigelli and L. Benini. 2017. Origami: A 803-GOp/s/W Convolutional Network Accelerator. *IEEE Transactions on Circuits and Systems for Video Technology* 27, 11 (Nov 2017), 2461–2475. https://doi.org/10.1109/TCSVT.2016.2592330

[15] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2018. Eyeriss v2: A Flexible and High-Performance Accelerator for Emerging Deep Neural Networks. *CoRR* abs/1807.07928 (2018). arXiv:1807.07928 http://arxiv.org/abs/1807.07928

[16] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory. *SIGARCH Comput. Archit. News* 44, 3 (June 2016), 27–39. https://doi.org/10.1145/3007787.3001140

[17] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014). arXiv:1406.1078 http://arxiv.org/abs/1406.1078

[18] F. Conti and L. Benini. 2015. A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters. In *2015 DATE*. 683–688. https://doi.org/10.7873/DATE.2015.0404

[19] Francesco Conti, Andrea Marongiu, and Luca Benini. 2013. Synthesis-friendly Techniques for Tightly-coupled Integration of Hardware Accelerators into Shared-memory Multi-core Clusters. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (Montreal, Quebec, Canada) *(CODES+ISSS '13)*. IEEE Press, Piscataway, NJ, USA, Article 5, 10 pages. http://dl.acm.org/citation.cfm?id=2555692.2555697

[20] J. Cui and Q. Qiu. 2016. Towards memristor based accelerator for sparse matrix vector multiplication. In *2016 ISCAS*. 121–124. https://doi.org/10.1109/ISCAS.2016.7527185

[21] Matthew D Zeiler and Rob Fergus. 2013. Visualizing and Understanding Convolutional Neural Networks. In *ECCV 2014, Part I, LNCS 8689*, Vol. 8689. https://doi.org/10.1007/978-3-319-10590-1_53

[22] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. 1990. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* 16, 1 (March 1990), 1–17. https://doi.org/10.1145/77626.79170

[23] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. 1988. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* 14, 1 (March 1988), 1–17. https://doi.org/10.1145/42288.42291

[24] Ronald G. Dreslinski, David Fick, Bharan Giridhar, Gyouho Kim, Sangwon Seo, Matthew Fojtik, Sudhir Satpathy, Yoonmyung Lee, Daeyeon Kim, Nurrachman Liu, Michael Wieckowski, Gregory Chen, Dennis Sylvester, David Blaauw, and Trevor Mudge. 2013. Centip3De: A Many-core Prototype Exploring 3D Integration and Near-threshold Computing. *Commun. ACM* 56, 11 (Nov. 2013), 97–104. https://doi.org/10.1145/2524713.2524725

[25] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *2015 ISCA*. 92–104. https://doi.org/10.1145/2749469.2750389

[26] Ahmed T. Elthakeb, Prannoy Pilligundla, Amir Yazdanbakhsh, Sean Kinzer, and Hadi Esmaeilzadeh. 2018. ReLeQ: A Reinforcement Learning Approach for Deep Quantization of Neural Networks. *CoRR* abs/1811.01704 (2018). arXiv:1811.01704 http://arxiv.org/abs/1811.01704

[27] E. Giacomin, T. Greenberg-Toledo, S. Kvatinsky, and P. Gaillardon. 2018. A Robust Digital RRAM-Based Convolutional Block for Low-Power Image Processing and Learning Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2018), 1–12. https://doi.org/10.1109/TCSI.2018.2872455

[28] Graham Gobieski, Nathan Beckmann, and Brandon Lucia. 2018. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. *CoRR* abs/1810.07751 (2018). arXiv:1810.07751 http://arxiv.org/abs/1810.07751

[29] Maximilian Golub, Guy Lemieux, and Mieszko Lis. 2018. DropBack: Continuous Pruning During Training. *CoRR* abs/1806.06949 (2018). arXiv:1806.06949 http://arxiv.org/abs/1806.06949

[30] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. 2018. UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 674–687. https://doi.org/10.1109/ISCA.2018.00062

[31] Parker Hill, Babak Zamirai, Shengshuo Lu, Yu-Wei Chao, Michael Laurenzano, Mehrzad Samadi, Marios C. Papaefthymiou, Scott A. Mahlke, Thomas F. Wenisch, Jia Deng, Lingjia Tang, and Jason Mars. 2018. Rethinking Numerical Representations for Deep Neural Networks. *CoRR* abs/1808.02513 (2018). arXiv:1808.02513 http://arxiv.org/abs/1808.02513

[32] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[33] C. Hsu, I. Wang, C. Lo, M. Chiang, W. Jang, C. Lin, and T. Hou. 2013. Self-rectifying bipolar TaOx/TiO2 RRAM with superior endurance over 1012 cycles for 3D high-density storage-class memory. In *2013 Symposium on VLSI Technology*. T166–T167.

[34] S. Huang, S. Xiao, and W. Feng. 2009. On the energy efficiency of graphics processing units for scientific computing. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–8. https://doi.org/10.1109/IPDPS.2009.5160980

[35] W. Huangfu, L. Xia, M. Cheng, X. Yin, T. Tang, B. Li, K. Chakrabarty, Y. Xie, Y. Wang, and H. Yang. 2017. Computation-oriented fault-tolerance schemes for RRAM computing systems. In *2017 ASP-DAC*. 794–799. https://doi.org/10.1109/ASPDAC.2017.7858421

[36] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). arXiv:1602.07360 http://arxiv.org/abs/1602.07360

[37] Mohsen Imani, Mohammad Samragh, Yeseong Kim, Saransh Gupta, Farinaz Koushanfar, and Tajana Rosing. 2018. RAPIDNN: In-Memory Deep Neural Network Acceleration Framework. *CoRR* abs/1806.05794 (2018). arXiv:1806.05794 http://arxiv.org/abs/1806.05794

[38] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, Raquel Urtasun, and Andreas Moshovos. 2018. Proteus: Exploiting precision variability in deep neural networks. *Parallel Comput.* 73 (2018), 40 – 51. https://doi.org/10.1016/j.parco.2017.05.003 Parallel Programming for Resilience and Energy Efficiency.

[39] K. Kasichayanula, D. Terpstra, P. Luszczek, S. Tomov, S. Moore, and G. D. Peterson. 2012. Power Aware Computing on GPUs. In *2012 Symposium on Application Accelerators in High Performance Computing*. 64–73. https://doi.org/10.1109/SAAHPC.2012.26

[40] Yulhwa Kim, Hyungjun Kim, and Jae-Joon Kim. 2018. Neural Network-Hardware Co-design for Scalable RRAM-based BNN Accelerators. *CoRR* abs/1811.02187 (2018). arXiv:1811.02187 http://arxiv.org/abs/1811.02187

[41] Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR* abs/1806.08342 (2018). arXiv:1806.08342 http://arxiv.org/abs/1806.08342

[42] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (May 2017), 84–90. https://doi.org/10.1145/3065386

[43] L. Kull, T. Toifl, M. Schmatz, P. A. Francese, C. Menolfi, M. Braendli, M. Kossel, T. Morf, T. M. Andersen, and Y. Leblebici. 2013. A 3.1mW 8b 1.2GS/s single-channel asynchronous SAR ADC with alternate comparators for enhanced speed in 32nm digital SOI CMOS. In *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*. 468–469. https://doi.org/10.1109/ISSCC.2013.6487818

[44] Andreas Kurth, Pirmin Vogel, Alessandro Capotondi, Andrea Marongiu, and Luca Benini. 2017. HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA. *CoRR* abs/1712.06497 (2017). arXiv:1712.06497 http://arxiv.org/abs/1712.06497

[45] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. *CoRR* abs/1801.06601 (2018). arXiv:1801.06601 http://arxiv.org/abs/1801.06601

[46] S. R. Lee, Y. Kim, M. Chang, K. M. Kim, C. B. Lee, J. H. Hur, G. Park, D. Lee, M. Lee, C. J. Kim, U. Chung, I. Yoo, and K. Kim. 2012. Multi-level switching of triple-layered TaOx RRAM with excellent reliability for storage class memory. In *2012 Symposium on VLSI Technology (VLSIT)*. 71–72. https://doi.org/10.1109/VLSIT.2012.6242466

[47] B. Li, L. Song, F. Chen, X. Qian, Y. Chen, and H. H. Li. 2018. ReRAM-based accelerator for deep learning. In *2018 DATE*. 815–820. https://doi.org/10.23919/DATE.2018.8342118

[48] S. Liao, Y. Xie, X. Lin, Y. Wang, M. Zhang, and B. Yuan. 2018. Reduced-Complexity Deep Neural Networks Design using Multi-Level Compression. *IEEE Transactions on Sustainable Computing* (2018), 1–1. https://doi.org/10.1109/TSUSC.2017.2710178

[49] Zhong Qiu Lin, Audrey G. Chung, and Alexander Wong. 2018. EdgeSpeechNets: Highly Efficient Deep Neural Networks for Speech Recognition on the Edge. *CoRR* abs/1810.08559 (2018).

[50] C. Liu, M. Hu, J. P. Strachan, and H. Li. 2017. Rescuing memristor-based neuromorphic design with high defects. In *2017 DAC*. 1–6. https://doi.org/10.1145/3061639.3062310

[51] Sparsh Mittal. 2018. A Survey of ReRAM-Based Architectures for Processing-In-Memory and Neural Networks. *Machine Learning and Knowledge Extraction* 1 (04 2018). https://doi.org/10.3390/make1010005

[52] Daisuke Miyashita, Shouhei Kousai, Tomoya Suzuki, and Jun Deguchi. 2016. Time-domain neural network: A 48.5 TSOp/s/W neuromorphic chip optimized for deep learning and CMOS technology. *2016 A-SSCC* (2016), 25–28.

[53] A. Nag, R. Balasubramonian, V. Srikumar, R. Walker, A. Shafiee, J. P. Strachan, and N. Muralimanohar. 2018. Newton: Gravitating Towards the Physical Limits of Crossbar Acceleration. *IEEE Micro* 38, 5 (Sep. 2018), 41–49. https://doi.org/10.1109/MM.2018.053631140

[54] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *2017 ISCA*. 27–40. https://doi.org/10.1145/3079856.3080254

[55] R. Pawlowski, E. Krimer, J. Crop, J. Postman, N. Moezzi-Madani, M. Erez, and P. Chiang. 2012. A 530mV 10-lane SIMD processor with variation resiliency in 45nm SOI. In *2012 ISSCC*. 492–494. https://doi.org/10.1109/ISSCC.2012.6177105

[56] A. Pullini, F. Conti, D. Rossi, I. Loi, M. Gautschi, and L. Benini. 2018. A Heterogeneous Multicore System on Chip for Energy Efficient Brain Inspired Computing. *TCAS II: Express Briefs* 65, 8 (Aug 2018), 1094–1098. https://doi.org/10.1109/TCSII.2017.2652982

[57] Ximing Qiao, Xiong Cao, Huanrui Yang, Linghao Song, and Hai Li. 2018. Atomlayer: A Universal reRAM-based CNN Accelerator with Atomic Layer Computation. In *Proceedings of DAC* (San Francisco, California) *(DAC '18)*. ACM, New York, NY, USA, Article 103, 6 pages. https://doi.org/10.1145/3195970.3195998

[58] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling Low-power, Highly-accurate Deep Neural Network Accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 267–278. https://doi.org/10.1109/ISCA.2016.32

[59] Davide Rossi, Igor Loi, Antonio Pullini, and Luca Benini. 2017. *Ultra-Low-Power Digital Architectures for the Internet of Things*. Springer International Publishing, Cham, 69–93. https://doi.org/10.1007/978-3-319-51482-6_3

[60] Davide Rossi, Antonio Pullini, Igor Loi, Michael Gautschi, Frank K. Gurkaynak, Andrea Bartolini, Philippe Flatresse, and Luca Benini. 2016. A 60 GOPS/W, -1.8V to 0.9V body bias ULP cluster in 28nm UTBB FD-SOI technology. *Solid-State Electronics* 117, C (2016), 170–184. https://doi.org/10.1016/j.sse.2015.11.015

[61] Pasquale D. Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. 2018. Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX. *IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference* (October 2018). https://doi.org/10.3929/ethz-b-000314427

[62] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A Unified Embedding for Face Recognition and Clustering. *CoRR* abs/1503.03832 (2015). arXiv:1503.03832 http://arxiv.org/abs/1503.03832

[63] S. Seo, R. G. Dreslinski, M. Woh, C. Chakrabarti, S. Mahlke, and T. Mudge. 2010. Diet SODA: A power-efficient processor for digital cameras. In *2010 ISLPED*. 79–84. https://doi.org/10.1145/1840845.1840862

[64] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *2016 ISCA*. 14–26. https://doi.org/10.1109/ISCA.2016.12

[65] Laurent Sifre and Stéphane Mallat. 2014. Rigid-Motion Scattering for Texture Classification. *CoRR* abs/1403.1687 (2014). arXiv:1403.1687 http://arxiv.org/abs/1403.1687

[66] J. Sim, J. Park, M. Kim, D. Bae, Y. Choi, and L. Kim. 2016. 14.6 A 1.42TOPS/W deep convolutional neural network recognition processor for intelligent IoE systems. In *2016 ISSCC*. 264–265. https://doi.org/10.1109/ISSCC.2016.7418008

[67] Aaron Stillmaker and Bevan M. Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration* 58 (2017), 74–81.

[68] Aaron Stillmaker, Zhibin Xiao, and Bevan Baas. 2011. *Toward More Accurate Scaling Estimates of CMOS Circuits from 180 nm to 22 nm.* Technical Report ECE-VCL-2011-4. VLSI Computation Lab, ECE Department, University of California,

Davis. http://www.ece.ucdavis.edu/cerl/techreports/2011-4/.

[69] C. Szegedy, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. 2015. Going deeper with convolutions. In *2015 IEEE CVPR*. 1–9. https://doi.org/10.1109/CVPR.2015.7298594

[70] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. 2014. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *CVPR*.

[71] R. Tang and J. Lin. 2018. Deep Residual Learning for Small-Footprint Keyword Spotting. In *2018 ICASSP*. 5484–5488. https://doi.org/10.1109/ICASSP.2018.8462688

[72] S. Tang, S. Yin, S. Zheng, P. Ouyang, F. Tu, L. Yao, J. Wu, W. Cheng, L. Liu, and S. Wei. 2017. AEPE: An area and power efficient RRAM crossbar-based accelerator for deep CNNs. In *2017 NVMSA*. 1–6. https://doi.org/10.1109/NVMSA.2017.8064475

[73] Chakkrit Termritthikun, Surachet Kanprachar, and Paisarn Muneesawang. 2018. NU-LiteNet: Mobile Landmark Recognition using Convolutional Neural Networks. *CoRR* abs/1810.01074 (2018). arXiv:1810.01074 http://arxiv.org/abs/1810.01074

[74] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3, Article 14 (June 2015), 33 pages. https://doi.org/10.1145/2764454

[75] Peiqi Wang, Yu Ji, Chi Hong, Yongqiang Lyu, Dongsheng Wang, and Yuan Xie. 2018. SNrram: An Efficient Sparse Neural Network Computation Architecture Based on Resistive Random-access Memory. In *Proceedings of DAC* (San Francisco, California) *(DAC '18)*. ACM, New York, NY, USA, Article 106, 6 pages. https://doi.org/10.1145/3195970.3196116

[76] S. Wang, D. Zhou, X. Han, and T. Yoshimura. 2017. Chain-NN: An energy-efficient 1D chain architecture for accelerating deep convolutional neural networks. In *DATE 2017*. 1032–1037. https://doi.org/10.23919/DATE.2017.7927142

[77] Andrew Waterman, Yunsup Lee, David A. Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. 2014. The RISC-V Instruction Set Manual.

[78] R. Clint Whaley and Antoine Petitet. 2005. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience* 35, 2 (February 2005), 101–121. http://www.cs.utsa.edu/~whaley/papers/spercw04.ps.

[79] Walt Woods and Christof Teuscher. 2017. Approximate vector matrix multiplication implementations for neuromorphic applications using memristive crossbars. *2017 NANOARCH* (2017), 103–108.

[80] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *2015 HPCA*. 476–488. https://doi.org/10.1109/HPCA.2015.7056056

[81] Mahmoud Zangeneh and Ajay Joshi. 2012. Performance and energy models for memristor-based 1T1R RRAM cell. (05 2012). https://doi.org/10.1145/2206781.2206786

[82] M. Zangeneh and A. Joshi. 2014. Design and Optimization of Nonvolatile Multibit 1T1R Resistive RAM. *TVLSI* 22, 8 (2014), 1815–1828.

[83] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *2018 MICRO*. 15–28. https://doi.org/10.1109/MICRO.2018.00011