The Bitlet Model
A Parameterized Analytical Model to Compare PIM and CPU Systems

RONNY RONEN, Technion - Israel Institute of Technology, Israel
ADI ELIAHU, Technion - Israel Institute of Technology, Israel
NATAN PELED, Technion - Israel Institute of Technology, Israel
KUNAL KORGAONKAR, Technion - Israel Institute of Technology, Israel
ANUPAM CHATTOPADHYAY, Nanyang Technological University, Singapore
SHAHAR KVATINSKY, Technion - Israel Institute of Technology, Israel

Nowadays, data-intensive applications are gaining popularity and, together with this trend, processing-in-memory (PIM)-based systems are being given more attention and have become more relevant. This paper describes an analytical modeling tool called Bitlet that can be used, in a parameterized fashion, to estimate the performance and the power of a PIM-based system and thereby assess the affinity of workloads for PIM as opposed to traditional computing. The tool uncovers interesting tradeoffs between, mainly, the PIM computation complexity (cycles required to perform a computation through PIM), the amount of memory used for PIM, the system memory bandwidth, and the data transfer size. Despite its simplicity, the model reveals new insights when applied on real-life examples. The model is demonstrated for several synthetic examples, and then applied to explore the influence of different parameters on two systems - IMAGING and FloatPIM. Based on the demonstrations, insights about PIM and its combination with CPU are concluded.

CCS Concepts: • **Hardware → Emerging architectures**; • **Computing methodologies → Model development and analysis**.

Additional Key Words and Phrases: Memristive Memory, Non-Volatile Memory, Processing in Memory, Analytical Models

ACM Reference Format:
Ronny Ronen, Adi Eliahu, Natan Peled, Kunal Korgaonkar, Anupam Chattopadhyay, and Shahar Kvatinsky. 2018. The Bitlet Model A Parameterized Analytical Model to Compare PIM and CPU Systems. *ACM J. Emerg. Technol. Comput. Syst.* 1, 1 (August 2018), 26 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Processing huge amounts of data on traditional von Neumann architectures involves many data transfers between the central processing unit (CPU) and the memory. These transfers degrade performance and consume energy [10, 12,

This work was supported by the European Research Council through the European Union's Horizon 2020 Research and Innovation Programme under Grant 757259 and by the Israel Science Foundation under Grant 1514/17.

Authors' addresses: Ronny Ronen, Technion - Israel Institute of Technology, Haifa, Israel; Adi Eliahu, Technion - Israel Institute of Technology, Haifa, Israel; Natan Peled, Technion - Israel Institute of Technology, Haifa, Israel; Kunal Korgaonkar, Technion - Israel Institute of Technology, Haifa, Israel; Anupam Chattopadhyay, Nanyang Technological University, Singapore, Singapore; Shahar Kvatinsky, Technion - Israel Institute of Technology, Haifa, Israel.

29, 31, 34, 35]. Enabled by emerging memory technologies, recent memristive processing-in-memory (PIM)[1] solutions show great potential in reducing costly data transfers by performing computations using individual memory cells [7, 23, 26, 32, 41]. Research in this area has led to better circuits and micro-architectures [5, 23, 24], as well as applications using this paradigm [14, 20].

PIM solutions have recently been integrated into application-specific [9] and general-purpose [18] architectures. General-purpose PIM-based architectures usually rely on memristive logic gates which are functionally complete sets to enable the execution of arbitrary logic functions within the memory. Different memristive logic gates have been designed and implemented, including MAGIC [23], IMPLY [8], and resistive majority [39].

Despite the recent resurgence of PIM, it is still very challenging to analyze and quantify the advantages or disadvantages of PIM solutions over other computing paradigms. We believe that a useful analytical modeling tool for PIM can play a crucial role in addressing this challenge. An analytical tool in this context has many potential uses, such as in (i) evaluation of applications mapped to PIM, (ii) comparison of PIM versus traditional architectures, and (iii) analysis of the implications of new memory technology trends on PIM.

Our Bitlet model (following [22]) is an analytical modeling tool that facilitates comparisons of PIM versus traditional CPU/GPU computing. The name Bitlet reflects PIM's unique bit-by-bit data element processing approach. The model is inspired by past successful analytical models for computing [11, 13, 15, 16, 40] and provides a simple operational view of PIM computations.

The main contributions of this work are:

- Presentation of use cases where using PIM has the potential to improve system performance by reducing data transfer in the system, and Quantification of the potential gain and the PIM computation cost of these use cases.
- Presentation of the Bitlet model, an analytical modeling tool that abstracts algorithmic, technological, as well as architectural machine parameters for PIM.
- Application of the Bitlet model on various workloads to illustrate how it can serve as a *litmus test* for workloads to assess their affinity on PIM as compared to the CPU.
- Delineation of the strengths and weaknesses of the new PIM paradigm as observed in a sensitivity study evaluating PIM performance and efficiency over various Bitlet model parameters.

The rest of the paper is organized as follows: Section 2 provides background on PIM. In Section 3, we describe the PIM potential use cases. In Section 4, we assess the performance of PIM, CPU and a PIM-CPU hybrid system. Section 5 discusses and compares the power and energy aspects of these systems. In Section 6, we present the Bitlet model and its ability to evaluate the potential of PIM and its applications. We conclude the paper in Section 7.

2 BACKGROUND

This section establishes the context of the Bitlet research. It provides information about current PIM developments, focusing on stateful logic-based PIM systems and outlining different methods that use stateful logic for logic execution within a memristive crossbar array.

2.1 Processing-In-Memory (PIM)

The majority of modern computer systems use the von Neumann architecture, in which there is a complete separation between processing units and data storage units. Nowadays, both units have reached a scaling barrier, and the data

[1] We refer to memristive stateful logic [33] as PIM, but the concepts and model may apply to other technologies as well.

processing performance is now limited mostly by the data transfer between these two units. The energy and delay associated with this data transfer are estimated to be several orders of magnitude higher than the cost of the computation itself [29, 30], and are even higher in data-intensive applications, which have become popular, *e.g.*, neural networks [36] and DNA sequencing [21]. This data transfer bottleneck is known as *the memory wall*.

The memory wall has raised the need to bridge the gap between where data resides and where it is processed. First, an approach called *processing-near-memory* was suggested, in which, computing units are placed close to or in the memory chip. Many architectures were designed using this method, *e.g.*, intelligent RAM (IRAM) [29], active pages [27], and 3D-stacked dynamic random access memory (DRAM) architectures [1]. However, this technique still requires data transfer between the memory cells and the computing units. Then, another approach, called PIM was suggested, in which, the memory cells also function as computation units. Various new and emerging memory technologies, *e.g.*, resistive random access memory (RRAM) [2], often referred to as memristors, have recently been explored. Memristors are new electrical components which can store two resistance values: R_{ON} and R_{OFF}, and therefore can function as memory elements. In addition, by applying voltage or passing current through memristors, they can change their resistance and therefore can also function as computation elements. These two characteristics make the memristor an attractive candidate for PIM.

2.2 Memristive Memory Architecture

Like other memory technologies, the memristive memory is usually organized in a hierarchical structure. Each RRAM chip is divided into *banks*. Each bank is comprised of *subarrays*, which are divided into different cell matrices (*MATs*). The MAT is a two-dimensional memristive crossbar array, where the data is stored and the in-situ computation is performed. The MAT crossbar memristive array consists of rows and columns. A memristive cell resides in each intersection of a row and column. Overall, the RRAM chip consists of many MATs, which can either share the same controller and perform similar calculations on different data, or have separate controllers for different groups of MATs and act independently.

2.3 Stateful Logic

Different logic families, which use memristive memory cells as building blocks to construct logic gates within the memory array, have been proposed in the literature. These families have been classified into various categories according to their characteristics: statefulnes, proximity of computation and flexibility [33]. In this paper, we focus on 'stateful logic' families, so we use the term PIM to refer specifically to stateful logic-based PIM, and we use the term *PIM technologies* to refer to different stateful logic families. A logic family is said to be stateful if the inputs and outputs of the logic gates in the family are represented by memristor resistance.

Several PIM technologies have been designed, including IMPLY [8] and MAGIC [23] gates. MAGIC gates have become a commonly used PIM technology. Figure 1(a) shows the MAGIC NOR logic gate structure, where the two input memristors are connected to an operating voltage, V_g, and the output memristor is grounded. Since MAGIC is a stateful logic family, the gate inputs and output are represented as memristor resistance. The input memristors are set with the input values of the logic gate and the output memristor is initialized at R_{ON}. The resistance of the output memristor changes during the execution according to the voltage divider rule, and switches when the voltage across it is higher than $\frac{V_g}{2}$. The same gate structure can be used to implement an OR logic gate, with minor modifications (the output memristor is initialized at R_{OFF} and a negative operating voltage V_g is applied) [17]. As depicted in Figures 1(b) and 1(c), a single MAGIC NOR gate can be mapped to a memristive crossbar array row (horizontal operation) or column

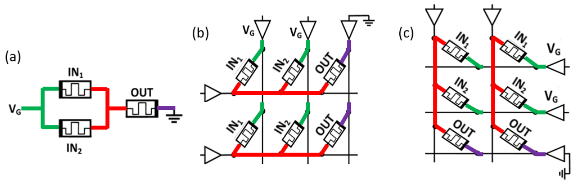


Fig. 1. MAGIC NOR gates. (a) MAGIC NOR gate schematic. (b) Two MAGIC NOR gates mapped to crossbar array rows, operated in parallel. (c) Two MAGIC NOR gates mapped to crossbar array columns, operated in parallel.

(vertical operation). Multiple MAGIC NOR gates can operate on different rows or columns concurrently, thus enabling massive parallelism.

2.4 Logic Execution within a Memristive Crossbar Array

A functionally complete memristive logic gate, *e.g.*, a MAGIC NOR gate, enables in-memory execution of any logic function. The in-memory execution is performed by a sequence of operations performed over several clock cycles. In each clock cycle, one operation can be performed on a single row or column, or on multiple rows or columns concurrently, if the data is row- or column-aligned. The execution of an arbitrary logic function with stateful logic has been widely explored in the literature [3, 4, 38, 42]. Many execution and mapping techniques first use a synthesis tool, which synthesizes the logic function and creates a netlist of logic gates. Then, each logic gate in the netlist is mapped to several cells in the memristive crossbar and operated in a specific clock cycle. Each technique maps the logic function according to its algorithm, based on different considerations, *e.g.*, latency, area or throughput optimization.

Many techniques use several rows or columns in the memristive crossbar array for the mapping [4, 6, 38] to reduce the number of clock cycles per a single function or to allow mapping of functions that are longer than the array row size. The unique characteristic of the crossbar array, which enables parallel execution of several logic gates in different rows or columns, combined with an efficient cell reuse feature that enables condensing long functions into short crossbar rows, renders single instruction multiple data (SIMD) operations attractive. In SIMD operations, the same function is executed simultaneously on multiple rows or columns. Executing logic in SIMD mode increases the computation throughput; therefore, by limiting the entire function mapping to a single row or column, the throughput can be substantially improved. This is applied in the SIMPLER [3] mapper. In this paper, we assume the logic function is mapped to a single row in the memristive crossbar and cloned to different rows for different data.

3 PIM USE CASES AND COMPUTATION PRINCIPLES

After presenting the motivation for PIM in the previous section, in this section, we describe potential use cases of PIM. We start with a high-level estimate of the potential benefits of reduced data transfer. Later, we define some computation principles, and using them, we assess the performance cost of PIM computing.

3.1 PIM Use Cases and Data Transfer Reduction

As stated in Section 2, the benefit of PIM comes mainly from the reduction in the amount of the data transferred between the memory and the CPU. If the saved time and energy due to the data transfer reduction is higher than the added cost of PIM processing, then PIM is beneficial. In this sub-section, we list PIM use cases that reduce data transfer and quantify this reduction.

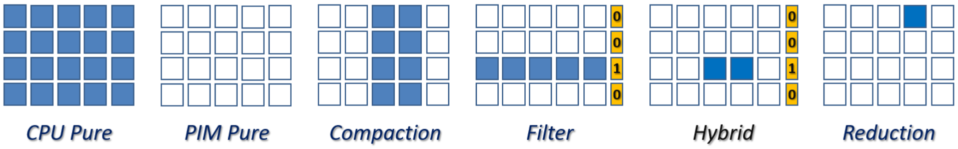


Fig. 2. Data size reduction illustration. Blue squares: data to be transferred, white: saved data transfer, yellow: bit vector of selected records to transfer.

For illustration, assume our data reflect a structured database in the memory that consists of N records, where each record is mapped into a single row in the memory. Each record consists of fields of varying data types. A certain compute task reads certain fields of each record, with overall size of S_i bits, and writes back S_o (potentially zero) bits as the output. Define $S = S_i + S_o$ as the total number of accessed bits per record. Traditional CPU-based computation consists of transferring $N \times S_i$ bits from memory to the CPU, performing the needed computations, and writing back $N \times S_o$ bits to the memory. In total, this computations requires transfer of $N \times S$ bits between the memory and the CPU. By performing all or part of the computations in memory, the total amount of data transfer can be reduced. This reduction is achieved by either reducing (or eliminating) the bits transferred per record (”*Compacting*”), and/or by reducing the number of transferred records (”*Filtering*”).

Several potential use cases follow, all of which differ in the way a task is split between PIM and CPU. In all cases, we assume that all records are appropriately aligned in the memory, so that PIM can perform the basic computations on all records concurrently (handling unaligned data is discussed later in Section 3.2). Figure 2 illustrates these use cases.

- **CPU Pure.** This is the baseline use case. No PIM is performed. All input and output data are transferred to CPU and back. The amount of data transferred is $N \times S$ bits.
- **PIM Pure.** In this extreme case, the entire computation is done in memory and no data is transferred. This kind of computation is done, for example, as a pre-processing stage in anticipation for future queries. See relevant examples under *PIM Compact* and *PIM filter* below.
- **PIM Compact.** Each record is processed in memory in order to reduce the amount of bits to be transferred to the CPU from each record. For example, each product record in a warehouse database contains 12 monthly shipment quantity fields. The application only needs the yearly quantity. Summing ("Compacting") these 12 elements into one reduces the amount of data transferred by 11 elements per record. Another example is an application that does not require the explicit shipping weight values recorded in the database, but just a short class tag (light, medium, heavy) instead. If the per-record amount of data is reduced from S to S_1 bits, then the overall reduction is $N \times (S - S_1)$ bits.
- **PIM Filter.** Each record is processed in memory in order to reduce the amount of records to be transferred to the CPU. This is a classical database query case. For example, A certain application looks for all shipments over $1M. Instead of passing all records to the CPU and check the condition in the CPU, the check is done in memory and only the records that pass the check ("Filtering") are transferred. If only N_1 out of N records of size S are selected, then the overall data transfer is reduced by $(N - N_1) \times S$ bits.

  Looking deeper, we need to take two more factors into account:

  (1) When the PIM does the filtering, the location of the selected records should also be transferred to the CPU, and the cost of transferring this information should be accounted for. Transferring the location can be done by either ($Filter_1$) passing a bit vector (N bits) or by ($Filter_2$) passing a list of indices of the selected records

($N_1 \times log_2(N)$ bits). The amount of the total data to be transferred is therefore $min(N \times 1, N_1 \times log_2(N))$. For simplicity, in this paper, we assume passing a bit vector ($Filter_1$). The overall cost of transferring both the data and the bit vector is $N_1 \times S + N$. The amount of saved data transfer relative to *CPU Pure* is $N \times S_1 - N$ bits.
(2) When filtering is done on the CPU only, data may be transferred twice. First, only a subset of the fields (the size of which is S_1) that are needed for the selection process are transferred, and only then, the selected records or a different subset of the records. In this *CPU Pure* case, the amount of transferred data is $N \times S_1 + N_1 \times S$.

- **PIM Hybrid.** This use case is a simple combination of applying both *PIM Compact* and *PIM filter*. The amount of data transferred depends on the method we use to pass the list of selected records, denoted above as $Filter_1$ or $Filter_2$. For example, when using $Filter_1$, the transferred data consists of N_1 records of size S_1 and a bit-vector of size N bits. That is $N_1 \times S_1 + N$.
- **PIM Reduction.** The reduction operator *"reduces the elements of a vector into a single result"*[2], *e.g.*, computes the sum, the minimum, or the maximum of a certain field in all records in the database. The size of the result may be equal to or larger than the original element size (*e.g.*, summing a million of 8-bits arbitrary numbers requires 28 bits). "Text book" reduction, referred to later as $Reduction_0$, replaces N elements of size S with a single element of size S_1 ($S_1 \geq S$), thus eliminating data transfer almost completely. A practical implementation, referred to later as $Reduction_1$, performs the reduction on each memory array (MAT) separately and passes all interim reduction results to the CPU for final reduction. In this case, the amount of transferred data is the product of the number of memory arrays used by the element size, *i.e.*, $\lceil \frac{N}{R} \rceil \times S_1$, where R is the number of records (rows) in a single MAT.

Table 1 summarizes all use cases along with the amount of transferred and saved data. In this table, N and N_1 reflect the overall and selected number of the transferred records. S and S_1 reflect the original and final size of the transferred records.

Table 1. PIM Use Cases Data Transfer Reduction

Use Case	Records	Size	Data Transferred	Data Transfer Reduction
$CPU\ Pure$	N	S	$N \times S$	0
$PIM\ Pure$	0	0	0	$N \times S$
$PIM\ Compact$	N	S_1	$N \times S_1$	$N \times (S - S_1)$
$PIM\ Filter^1$	N_1	S	$N_1 \times S + N$	$N \times S_1 - N$
$PIM\ Filter^2$	N_1	S	$N_1 \times (S + log_2(N))$	$N \times S - N_1 \times (S + log_2(N))$
$PIM\ Hybrid$	N_1	S_1	$N_1 \times S_1 + N$	$N \times (S - 1) - N_1 \times S_1$
$PIM\ Reduction_0$	1	S_1	$1 \times S_1$	$N \times S - S_1$
$PIM\ Reduction_1$	$N_1 = \lceil \frac{N}{R} \rceil$	S_1	$N_1 \times S_1$	$N \times S - N_1 \times S_1$

N/N_1: overall/selected number of records; S/S_1: original/final size of records.
$Filter_1/Filter_2$: bit-vector/list of indices; $Reduction_0/Reduction_1$: all records/per MAT

3.2 PIM Computation Principles

Stateful logic-based PIM (or just *PIM* throughout this paper) computation provides very high parallelism. Assuming that the structured database example above (Section 3.1) where each record is mapped into a single memory row, indeed, PIM can generate only a single bit result per record per memory cycle (*e.g.*, a single NOR, IMPLY, AND, *etc.,* based on the PIM technology in use), so the sequence needed to carry out a certain computation may be rather long. Nevertheless,

[2] https://en.wikipedia.org/wiki/Reduction_Operator

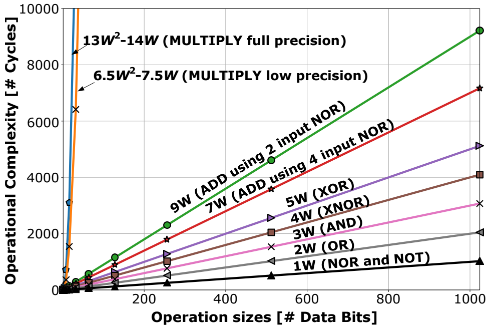


Fig. 3. PIM operation complexity in cycles for different types of operations and data sizes.

PIM can compute many properly aligned records during the same cycle. Proper alignment means that all the input cells and the output cell of all records occupy the same column in all participating memory rows (records), or, inversely, the input cells and the output cell occupy the same rows in all participating columns.

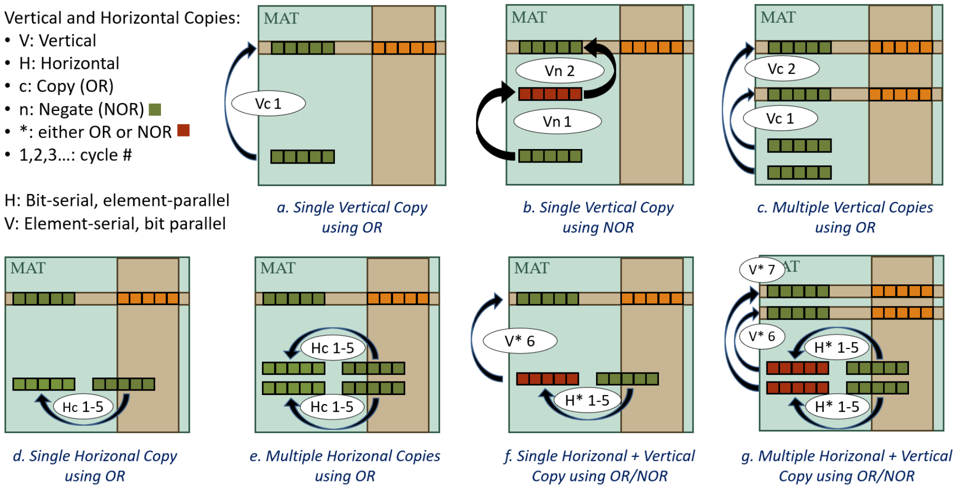
PIM can perform the same operation on many independent rows, and many MATs, simultaneously. However, performing operations involving computations between rows (*e.g.*, shift or reduction) or in-row copy of an element with a different alignment in each row, has a limited parallelism. Such copies can be done in parallel among MATs, but within a MAT, are performed mostly serially. When the data in a MAT are aligned, operations can be done in parallel (as demonstrated in Figures 1(b) and 1(c) for row-aligned and column-aligned operations, respectively). However, operations on unaligned data cannot be done concurrently, as further elaborated in Section 3.2.

To quantify PIM performance, we first separate the computation task into two steps: *Operation* and *Placement and Alignment*. Below, we assess the complexity of each of these steps. For simplicity, we assume N computations done on R rows and $MATs$ memory arrays, *i.e.*, $N = R \times MATs$.

Operation Complexity (*OC*). As described in Section 2.3, PIM computations are carried out as a series of basic operations, applied to the memory cells of a row inside a memristive memory array. While each row is processed bit-by-bit, the effective throughput of PIM is increased by the inherent parallelism achieved by simultaneous processing of multiple rows inside a memory array and of multiple memory arrays in the system memory. We assume the same computations (*i.e.*, individual operations) applied to a row are also applied in parallel in every cycle across all the rows (R) of a memory array.

We define **Operation Complexity** (*OC*) for a given operation type and data size, as the number of cycles required to process the corresponding data. Figure 3 shows how the input data length (W) affects the computing cycles for PIM-based processing. The figure shows that this number is affected by both the data size, as well as operation types (different operations follow a different curve on the graph). In many cases, *OC* is linear with the data size, for example, in a MAGIC NOR-based PIM, W-bit AND requires $3W$ cycles (*e.g.*, for W=16 bits AND takes 16x3 = 48 cycles), while ADD requires $9W$ cycles[3]. Some operations, however, are not linear, *e.g.*, full precision MULTIPLY $W \times W \longrightarrow 2W$ bits requires $13W^2 - 14W$ cycles [14] or approximately $12.5W^2$ cycles, while low precision MULTIPLY $W \times W \longrightarrow W$ bits requires about half the number of cycles, or approximately $6.25W^2$ cycles. The specific Operation Complexity behavior depends on the PIM technology, but the principle is similar.

[3] ADD can be improved to $7w$ cycles using an algorithmic optimization that uses four-input NOR instead of two-input NOR.



a. Single Vertical Copy using OR

b. Single Vertical Copy using NOR

c. Multiple Vertical Copies using OR

d. Single Horizonal Copy using OR

e. Multiple Horizonal Copies using OR

f. Single Horizonal + Vertical Copy using OR/NOR

g. Multiple Horizonal + Vertical Copy using OR/NOR

Fig. 4. Horizontal Copies (*HCOPY*) and Vertical Copies (*VCOPY*) using PIM.

Placement and Alignment Complexity (*PAC*). PIM imposes certain constraints on data alignment and placement [37]. To align the data for subsequent row-parallel operations, a series of data alignment and placement steps, consisting of copying data from one place to another, may be needed. The number of cycles needed to perform these additional copy steps is captured by the placement and alignment complexity parameter, denoted as *PAC*. Currently, for simplicity, we consider only the cost of intra-MAT data copying, and ignore the cost of inter-MAT data copying and assume that multiple memory arrays continue to operate in parallel and independently. Refining the model to account for inter-MAT data copying will be considered in the future (see Section 6.5).

The PAC cycles required to copy the data in a memory array to the desired locations can be broken down into a series of horizontal row-parallel, copies (*HCOPY*), and vertical column-parallel, copies (*VCOPY*). Figure 4 shows examples of VCOPY and HCOPY. The HCOPYs are performed bit-by-bit for a given data element, and hence, their cycle count is typically proportional to the size of the data element involved. When the input and output alignment is the same for all data elements, all bits with the same column index in all rows are copied in parallel. When the involved data elements across different rows are not aligned, separate HCOPYs are performed individually for each data element, thus requiring additional cycles. A VCOPY for a given data element, on the other hand, can be done in parallel on all the bits in the element, which are in the same row. However, each row within a MAT has to be vertically copied separately, in a serial manner.

The number of cycles it takes to perform a single bit copy (either HCOPY or VCOPY) depends on the PIM technology used. For example, MAGIC OR-based PIM technology ([17]) supports logic OR as a basic operation, allowing a 1-cycle bit copy (see Figure 4a, 4c, 4d, and 4e). PIM Technologies that do not support a 1-cycle bit copy (*e.g.*, MAGIC NOR-based PIM technology), have to execute two consecutive NOT operations that take two cycles to copy a single bit (Figure 4b). However, copying a single bit using a sequence of a HCOPY operation followed by a VCOPY operation can be implemented as two consecutive OR or NOT operations that take two cycles regardless of the PIM technology used (Figure 4f and 4g).

We define **Computation complexity** (*CC*) as the number of cycles required to fully process the corresponding data. *CC* equals to the sum of *OC* and *PAC*.

Below are examples of PIM *CC* cycles. We use the terms *Simple* and *Complex* to refer to the number of placements and alignments needed to align all elements. Simple means that all input locations are fully aligned among themselves, but not with their destination, while *Complex* means that input locations are not aligned among themselves.

- **Parallel aligned operation.** Adding two vectors, A and B, into vector C, where A_i, B_i and C_i are in row i. The size of each element is W-bits. A MAGIC NOR-based full adder operation takes $o(=9)$ cycles. Adding two W-bit elements in a single row takes $OC = o \times W$ cycles. At the same OC cycles, one can add either one element or millions of elements. Since there are no vertical or horizontal copies, the CC equals the OC. The above-mentioned PIM Compact, PIM Filter and PIM Hybrid use cases are usually implemented as parallel aligned operations.
- **Simple placement and alignment copies.** We want to perform a shifted vector copy, *i.e.*, copying vector A into vector B such that $B_{i-1} \leftarrow A_i$.[4] The size of each element is W-bit. With stateful logic, the naive way of making such a copy for a single element is by a sequence of $HCOPY$ operations followed by $VCOPY$ operations. For a given single element in a row A_i, first, copy all W bits of A_i in parallel, so $B_i \leftarrow A_i$, then, copy $B_{i-1} \leftarrow B_i$. Copying W-bits in a single row takes W cycles. As in the above simple aligned case, in the same W cycles, one can copy either one element or many elements. However, in this case, we also need to copy the result elements from one row to the adjacent one above. Copying W-bits between two rows takes a single cycle, as all W bits can be copied from one row to another in parallel. But, copying all rows is a serial operation, as it must be done separately for each row in the MAT. Hence, if the memory array contains R rows, the entire copy task will take $CC = (W + R)$ cycles. Still, these operations can be done in parallel on all the MATs in the system. Hence, copying all N elements can be completed in the same $(W + R)$ cycles.
- **Simple unaligned operation.** The time to perform a combination of the above two operations, *i.e.*, $C_{i-1} \leftarrow A_i + B_i$, is the sum of both computations, that is $CC = (OC + W + R)$ cycles.
- **Complex placement and alignment.** We want to gather elements from random memory locations into a row-aligned vector B, that is, all B elements occupy the same columns. Assume the worst case where all elements have to be horizontally and vertically copied to reach their desired location, as described above for *simple placement and alignment*. To accomplish this, we need to do W horizontal 1-bit copies and a one parallel W-bit copy for each element, totalling overall $CC = (W + 1) \times R$ cycles.
- **Complex unaligned operation.** Perform a *complex placement and alignment* followed by a *parallel aligned operation*, takes the sum of both computations, that is, $CC = (OC + (W + 1) \times R)$ cycles.
- **Reduction.** We look at a classical reduction where the reduction operation is both commutative and associative (*e.g.*, a sum, a minimum, or a maximum of a vector). For example, we want to sum a vector A where each element and the final sum are of size W, *i.e.*, $S = \sum_{i=1}^{N} A_i$. The idea is to first reduce all elements in each MAT into a single value separately, but in parallel, and then perform the reduction on all interim results. There are several ways to perform a reduction, the efficiency of which depends on the number of elements and the actual operation. We use the tree-like reduction[5], which is a phased process, in which at the beginning of each phase, we start with k elements ($k = R$, the number of rows, in the first phase), pair them into $k/2$ groups, perform all $k/2$ additions, and start a new phase with the $k/2$-generated numbers. For R elements, we need $ph = \lceil \log_2(R) \rceil$ phases. Each phase consists of one parallel (horizontal) copy of W bits, followed by $k/2$ serial (vertical) copies, and ending with one parallel operation (in our case, W-bit add). The total number of vertical copies is $R - 1$. Overall, the full reduction of a single MAT in all phases takes $CC = (ph \times (OC + W) + (R - 1))$ cycles. The reduction is done on all involved

[4] We ignore the elements of B that are last in each MAT.
[5] https://en.wikipedia.org/wiki/Graph_reduction

MATs in parallel, producing a single result per MAT. Later, all per MAT interim results are copied into fewer MATs and the process continues recursively over $\frac{\log_2(N)}{\log_2(R)}$ steps. Copying all interim results into fewer MATs and using PIM on a smaller number of MATs is inefficient as it involves serial inter-MAT copies and low-parallel PIM computations. Therefore, for higher efficiency, after the first reduction step is done using PIM, all interim results are passed to the CPU for the final reduction, denoted as $Reduction_1$ in Section 3.

Table 2. PIM Computation Cycles for Aligned and Unaligned Computations

Computation type	Operate *Row* *Parallel*	HCOPY *Row* *Parallel*	VCOPY *Row* *Serial*	Total	Approximation
Parallel Operation	OC	-	-	OC	OC
Simple Placement & Alignment	-	W	R	$W + R$	R
Simple Unaligned Operation	OC	W	R	$OC + W + R$	$OC + R$
Complex Placement & Alignment	-	$R \times W$	R	$(W + 1) \times R$	$W \times R$
Complex Unaligned Operation	OC	$R \times W$	R	$OC + (W + 1) \times R$	$OC + W \times R$
Reduction$_1$	$ph \times OC$	$ph \times W$	$R - 1$	$ph \times (OC + W) + (R - 1)$	$ph \times OC + R$

OC: Operation Complexity, W: Width of element, R: Number of rows, ph: Number of reduction phases.

Table 2 summarizes the computation complexity in cycles of various PIM computation types (ignoring inter-MAT copies, as mentioned above). Usually, $OC >> W$ and $R >> W$, so $OC \pm W$ is approximately OC, and $R \pm 1$ and $R \pm W$ are approximately R. The last column in the table reflects this approximation. The approximation column hints to where most cycles go, depending on OC, R and W. Parallel operations depend on OC only, and are independent of R, the number of elements (rows). When placement and alignment take place, there is a serial part which depends on R and is a potential cause for computation slowdown.

4 PIM AND CPU PERFORMANCE

In the previous section the PIM use cases and computation complexity were introduced. In this section, we device the actual performance equations of PIM, CPU, and combined PIM+CPU systems.

4.1 PIM Throughput

CC represents the time it takes to perform a certain computation, similar to the latency of an instruction in a compute system. However, due to the varying parallelism within the PIM system, CC does not directly reflect the PIM system performance. To evaluate the performance of a PIM system, we need to find its system throughput, which is defined as the the number of computations performed within a time unit. Common examples are Operations Per Second (OPS) or Giga Operations per Second (GOPS). For a PIM Pure case, when completing N computations takeing T_{PIM} cycles, the PIM throughput TP_{PIM} is:

$$TP_{PIM} = \frac{N}{T_{PIM}}. \tag{1}$$

To determine T_{PIM}, we obtain the PIM CC, and multiply it by the PIM cycle time (CT). CT depends on the specific PIM technology used. To compute CC, we use the equations in Table 2. The number of computations N is the total number of elements participating in the process. When single-row based computing is used, this number is the number of all participating rows, which is the product of the number of rows within a MAT with the number of MATs, that is, $N = MATs \times R$. The PIM throughput is therefore

$$TP_{PIM} = \frac{MATs \times R}{CC \times CT}. \tag{2}$$

For example, consider the *Simple unaligned operation* case for computing shifted vector-add, $C_{i-1} \longleftarrow A_i + B_i$. Assuming $o = 9$ cycles (1-bit add), element size $W = 16$ bits, $R = 1024$ rows, and $MATs = 1024$ memory arrays, then $N = 1M$ elements. $OC = 9 \times 16 = 144$ cycles. The number of cycles to compute R elements also equals the time to compute N elements and is $CC = OC + R$ or $144 + 512 = 656$ cycles. The PIM throughput per cycle is $\frac{N}{CC} = \frac{1024 \times 1024}{656} = 1598$ computations per cycle. The throughput is $\frac{1598}{CT}$ computations per time unit. We can derive the throughput per second for a specific cycle time. For example, for a CT of 10ns, the PIM throughput is $\frac{1598}{10^{-8}} = 159.8 \times 10^9$ OPS$\approx$ 160 GOPS.

In the following sections, we explain the CPU Pure performance and throughput and delve deeper into the overall throughput computation when both PIM and CPU participate in a computation.

4.2 CPU Computation and Throughput

Performing a computation on the CPU involves moving data between the memory and the CPU (*Data Transfer*), and performing the actual computations (*e.g.*, ALU Operations) within the CPU core (*CPU Core*). Usually, on the CPU side, Data Transfer and CPU Core Operations can overlap, so the overall CPU throughput TP_{CPU} is the minimum between the data transfer throughput and the CPU Core Throughput.

Using PIM to accelerate a workload is only justified when the the workload performance bottleneck is the data transfer between the memory and the CPU rather than the CPU core operation. When the CPU core is the bottleneck, PIM cannot help. In such PIM-relevant workloads, the overall CPU throughput TP_{CPU} is solely determined by the data transfer throughput.

The data transfer throughput depends on the memory to CPU bandwidth and the amount of data transferred per computation. We define BW as the memory to CPU bandwidth in bits per second (bps), and DIO ($DATA\ IO$) as the number of bits transferred for each computation. That is:

$$TP_{CPU} = \frac{BW}{DIO}. \tag{3}$$

We demonstrate the data transfer throughput using, again, the shifted 16-bit vector-add example. In Table 3, we present three interesting cases, differing in their DIO size. (a) **CPU Pure.** The two inputs and the output are transferred between the memory and the CPU ($DIO = 48$). (b) **Inputs only.** Same as CPU Pure, except that only the inputs are transferred to the CPU; no output result is written back to memory ($DIO = 32$). (c) **Compaction.** Where PIM performs the add operation and passes only the output data to the CPU for further processing ($DIO = 16$). We use the same data bus bandwidth, $BW = 1000$ GOPS, for all the three cases. Note that the data transfer throughput depends only on the data sizes, it is independent of the operation type. The throughput numbers in the table reflect any binary 16-bit operation, either simple as OR or complex as divide. The table hints to the potential gain that PIM opens by reducing the amount of data transfer between the memory and CPU. If PIM throughput is sufficiently high, the data transfer reduction may compensate for the additional PIM computations and the combined PIM+CPU system throughput may exceed the throughput of a system using the CPU only with PIM.

Special care must be taken when determining DIO for the PIM Filter and PIM Reduction cases, since only a subset of the records is transferred to the CPU. Note that the DIO parameter reflects the number of data bits transferred per accomplished computation, even though the data for some computations was not eventually transferred. In these cases, the DIO should be set as the total number of transferred data bits divided by the number of computations done in the system. For example, assume a filter, where we process N records of size S, and pass only $M = N \times p$ of them ($p < 1$). The DIO, in case we use a bit-vector to identify chosen records, is $\frac{(N\times p\times S)+N}{N} = (S \times p) + 1$. *e.g.*, if $S = 200$ and $p = 1\%$, DIO is $200 \times 0.01 + 1 = 2 + 1 = 3$ bits. That is, the amount of data transfer per computation was went from 200 to 3 bits per computation, *i.e.*, 67× reduction. The data transfer throughput for the filter case is presented in Table 3.

Table 3. Data Transfer Throughput

Computation type	Bandwidth (BW) [Gbps]	DataIO (DIO) [bits]	Data Transfer Throughput (TP_{CPU}) [GOPS]
CPU Pure	1000	48	20.8
Inputs Only	1000	32	31.3
Compaction	1000	16	62.5
Filter (200 bit, 1%)	1000	3	333.3

4.3 Combined PIM and CPU Throughput

Due to the parallel nature of the PIM, in a system which combines both PIM and CPU, PIM computations and data transfer cannot overlap. Therefore, completing N computations takes T_{PIM} PIM time and T_{CPU} data transfer time, and the combined throughput $TP_{Combined}$ is, by definition:

$$TP_{Combined} = \frac{N}{T_{PIM}+T_{CPU}}. \tag{4}$$

Fortunately, computing the combined throughput $TP_{Combined}$ does not require knowing the values of T_{PIM} and T_{CPU}. $TP_{Combined}$ can be computed using the throughput values of its components, TP_{PIM} and TP_{CPU}, as follows:

$$TP_{Combined} = \frac{N}{T_{PIM}+T_{CPU}} = \frac{1}{\frac{T_{PIM}}{N}+\frac{T_{CPU}}{N}} = \frac{1}{\frac{1}{TP_{PIM}}+\frac{1}{TP_{CPU}}}. \tag{5}$$

Since the PIM and CPU operations do not overlap, the combined throughput is always lower than the throughout of each component for the pure cases with the same parameters. For example, in the *Simple unaligned operation* case above, when computing a 16-bit shifted vector-add, *i.e.*, $C_{i-1} \longleftarrow A_i + B_i$, we do the vector-add in PIM, and transfer the 16-bit result vector to the CPU (for additional processing). We have already shown that, for the parameters we use, the PIM Throughput is $TP_{PIM} = 160$ GOPS, and the data transfer throughput is $TP_{CPU} = 62.5$ GOPS. Using Eq. (5), the combined throughput $TP_{Combined} = \frac{1}{\frac{1}{160\times10^9}+\frac{1}{62.5\times10^9}} = 44.9 \times 10^9 = 44.9$ GOPS, which is indeed lower than 160 and 62.5 GOPS. However, this combined throughput is higher than that of the CPU Pure throughput using higher DIO=32 or DIO=48 (31.3 or 20.8 GOPS) presented in the previous subsection. Of course, these results depend on the specific parameters used here. A comprehensive analysis of the performance sensitivity is described in Section 6.2.

5 POWER AND ENERGY

When evaluating the power and energy aspects of a system, we examine two factors:

- **Energy per computation.** The energy needed to accomplish a single computation. This energy is determined by the amount of work to be done (*e.g.*, number of basic operations) and the energy per operation. Different algorithms may produce different operation sequences thus affecting the amount of work to be done. Physical characteristics of the system affect the energy per operation. Energy per Computation is a measure of system efficiency. A system configuration that consumes less energy per a given computation is considered more efficient. For convenience, we generally use Energy Per Giga Computations. Energy is measured in Joules.

- **Power.** The power consumed while performing a computation. The maximum allowed power is usually determined by physical constrains like power supply and thermal restrictions, and may limit system performance. It is worth noting that the high parallel computation of PIM causes the memory system to consume much more power when in PIM mode than when in standard memory load/store mode. Power is measured in Watts, which are Joules per Second.

In this section we evaluate the PIM, the CPU and the combined system power and energy per computation and how it may impact system performance. For the sake of this coarse-grained analysis, we consider dynamic power only and ignore power management and dynamic voltage scaling.

5.1 PIM Power and Energy

Every PIM operation consumes energy. For simplicity, we assume that in every PIM cycle, the switching of a single cell consumes a certain energy $Ebit_{PIM}$. This amount of energy is consumed separately for each participating row in each MAT. The PIM energy per computation EPC_{PIM} is the product of $Ebit_{PIM}$ by the number of cycles CC. The PIM power P_{PIM} is the product of the energy per computation EPC_{PIM} by the PIM throughput TP_{PIM} (see Section 4.1).

$$EPC_{PIM} = Ebit_{PIM} \times CC, \tag{6}$$

$$P_{PIM} = EPC_{PIM} \times TP_{PIM} = (Ebit_{PIM} \times CC) \times \frac{MATs \times R}{CC \times CT} = \frac{Ebit_{PIM} \times R \times MATs}{CT}. \tag{7}$$

5.2 CPU Power and Energy

Here we compute the CPU energy per computation EPC_{CPU} and the power P_{CPU}. As in the performance model, we ignore the actual CPU Core operations and consider only the data transfer power and energy. Assume that transferring a single bit of data consumes $Ebit_{CPU}$. Hence, the CPU energy per computation EPC_{CPU} is the product of $Ebit_{CPU}$ by the number of bits per computation DIO. The CPU power P_{CPU} is simply the product of the energy per computation EPC_{CPU} with the CPU Throughput TP_{CPU}. When the memory to CPU bus is not idle, the CPU power P_{CPU} is equal to the product of the energy per bit $Ebit_{CPU}$ with the number of bits per second, which is the memory to CPU bandwidth BW.

$$EPC_{CPU} = Ebit_{CPU} \times DIO, \tag{8}$$

$$P_{CPU} = EPC_{CPU} \times TP_{CPU} = Ebit_{CPU} \times DIO \times \frac{BW}{DIO} = Ebit_{CPU} \times BW. \tag{9}$$

If the bus is busy only part of time, the CPU power P_{CPU} should be multiplied by the the relative time the bus is busy, that is, the bus duty cycle,

5.3 Combined PIM and CPU Power and Energy

When a task is split between PIM and CPU, we treat them as if part of each computation is partly done on the PIM and partly on the CPU (see Section 4.3). The combined energy per computation $EPC_{Combined}$ is the sum of the PIM energy per computation EPC_{PIM} and the CPU energy per computation EPC_{CPU}. The overall system power is the product of the combined energy per computation and the combined system throughput:

$$EPC_{Combined} = EPC_{PIM} + EPC_{CPU} = \frac{P_{PIM}}{TP_{PIM}} + \frac{P_{CPU}}{TP_{CPU}}, \tag{10}$$

$$P_{Combined} = EPC_{Combined} \times TP_{Combined} = (\frac{P_{PIM}}{TP_{PIM}} + \frac{P_{CPU}}{TP_{CPU}}) \times TP_{Combined}. \tag{11}$$

Since PIM and CPU computations do not overlap, their duty cycle is less than 100%. Therefore, the PIM power in the combined PIM+CPU system is lower than the maximum PIM Power in a Pure PIM configuration. Similarly, the CPU Power in the combined PIM+CPU system is lower than the maximum CPU Power.

In order to compare energy per computation between different configurations, we use the relevant EPC values, computed by dividing the power of the relevant configuration by its throughput. That is:

$$EPC_{PIM} = \frac{P_{PIM}}{TP_{PIM}};\ EPC_{CPU} = \frac{P_{CPU}}{TP_{CPU}};\ EPC_{Combined} = \frac{P_{Combined}}{TP_{Combined}}. \tag{12}$$

The following example summarizes the entire power and energy story. Assume, again, the above shifted vector-add example using the same PIM and CPU parameters. In addition, we use $Ebit_{PIM}$ = 0.1pJ [25] and $Ebit_{CPU}$ = 15pJ [28]. The PIM Pure throughput is 160 GOPS (see Section 4.1) and the PIM Pure power is $P_{PIM} = \frac{Ebit_{PIM} \times R \times MATs}{CT} = \frac{0.1 * 10^{-12} \times 1024 \times 1024}{10^{-8}}$ = 10.5W. The CPU Pure throughput (using BW = 1000 Gpbs) is 20.8 (or 62.5) GOPS for 48 (or 16) bit DIO (see Section 4.2). The CPU Pure Power is $P_{CPU} = Ebit_{CPU} \times BW$ =15*$10^{-12} \times 10^{12}$ = 15W. A combined PIM+CPU system will exhibit throughput of $TP_{Combined}$ = 44.9 GOPS and power $P_{Combined} = (\frac{P_{PIM}}{TP_{PIM}} + \frac{P_{CPU}}{TP_{CPU}}) \times TP_{Combined} = (\frac{10.5}{160 \times 10^9} + \frac{15}{62.5 \times 10^9}) \times (44.9 \times 10^9)$ = 13.7W.

Again, these results depend on the specific parameters in use. However, they demonstrate a case where, with PIM, not only the system throughput went up, but, at the same time, the system power decreased. When execution time and power consumption go down, energy goes down as well. In our example, $ECP_{CPU} = \frac{15}{20.8 \times 10^9} = \frac{0.72}{10^9}$ J/OP = 0.72 J/GOP, and $ECP_{Combined} = \frac{13.7}{44.9 \times 10^9} = \frac{0.31}{10^9}$ J/OP = 0.31 J/GOP.

5.4 Power-Constrained Operation

Occasionally, a system, or its components, may be power-constrained. For example, using too many MATs in parallel, or fully utilizing the memory bus may exceed the maximum allowed system or component *thermal design power*[6] (*TDP*). For example, the PIM power P_{PIM} must never exceed TDP_{PIM}. When a system or a component exceeds its TDP, it has to be slowed down to reduce its throughput and hence, its power consumption. For example, a PIM system throughput can be reduced by activating fewer MATs or rows in each cycle, or increasing the cycle time, or combination of both. CPU power can be reduced by forcing idle time on the memory bus to limit its bandwidth (*i.e.*, *"throttling"*).

6 THE BITLET MODEL - PUTTING IT ALL TOGETHER

So far, we have established the main principles of the PIM and CPU performance. In this section, we first present the Bitlet model itself, basically summarizing the relevant parameters and equations to compute the PIM, CPU, and combined performance in terms of throughput. Then, we demonstrate the application of the model to evaluate the potential benefit of PIM for various use cases. We conclude with a sensitivity analysis studying the interplay and impact of the various parameters on the PIM and CPU performance and power.

[6] en.wikipedia.org/wiki/Thermal_design_power

Table 4. *Bitlet Model Parameters.*

Parameter name	Notation	Value(s)	Type
PIM operation complexity	OC	1 - 32k cycles	Algorithmic
PIM placement and alignment complexity	PAC	0 - 32k cycles	Algorithmic
PIM computational complexity	$CC = OC{+}PAC$	1 - 64k cycles	Algorithmic
PIM cycle time	CT	10 ns [25]	Technological
PIM array dimensions (rows × columns)	$R \times C$	1024 x 1024	Technological
PIM array count	$MATs$	1k - 16k	Architectural
PIM energy for operation (OC=1) per bit	$Ebit_{PIM}$	0.1pJ [25]	Technological
CPU memory bandwidth	BW	1 to 16 Tbps	Algorithmic
CPU data in-out bits	DIO	16, 24, 32, 48, ... bits	Algorithmic
CPU energy per bit transfer	$Ebit_{CPU}$	15pJ [28]	Technological

Table 5. *Bitlet model Equations*

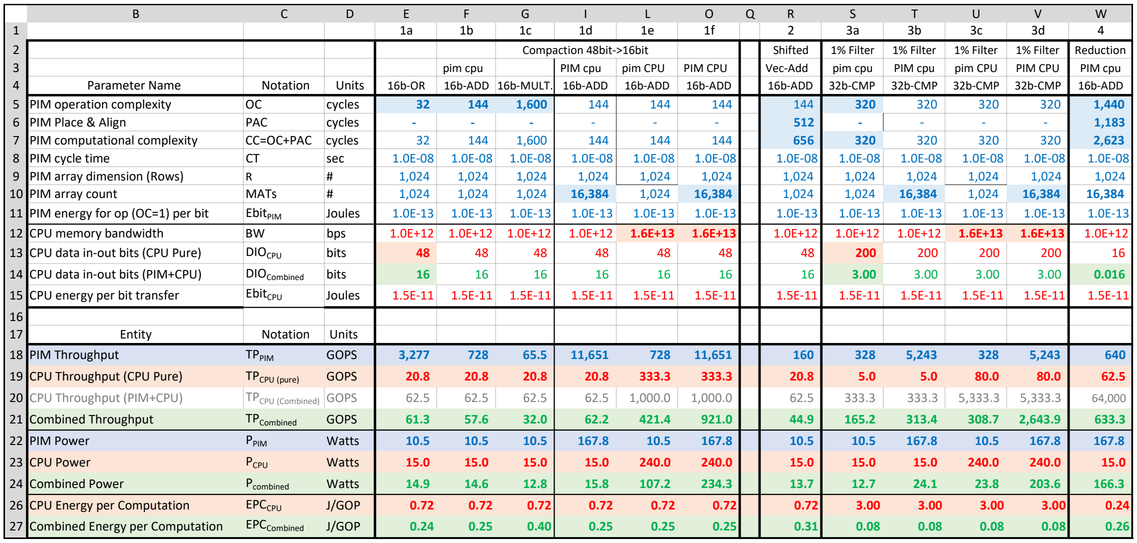
Entity	Equation	Units
PIM Throughput	$TP_{PIM} = \frac{R \times MATs}{CC \times CT}$	GOPS
CPU Throughput	$TP_{CPU} = \frac{BW}{DIO}$	GOPS
Combined Throughput	$TP_{Combined} = \frac{1}{\frac{1}{TP_{PIM}} + \frac{1}{TP_{CPU}}}$	GOPS
PIM Power	$P_{PIM} = \frac{Ebit_{PIM} \times R \times MATs}{CT}$	Watts
CPU Power	$P_{CPU} = Ebit_{CPU} \times BW$	Watts
Combined Power	$P_{Combined} = (\frac{P_{PIM}}{TP_{PIM}} + \frac{P_{CPU}}{TP_{CPU}}) \times TP_{Combined}$	Watts
PIM Energy per Computation	$EPC_{PIM} = \frac{P_{PIM}}{TP_{PIM}}$	J/GOP
CPU Energy per Computation	$EPC_{CPU} = \frac{P_{CPU}}{TP_{CPU}}$	J/GOP
Combined Energy per Computation	$EPC_{Combined} = \frac{P_{Combined}}{TP_{Combined}}$	J/GOP

6.1 The Bitlet Model Implementation

The Bitlet model consists of ten parameters and nine equations that define the throughput, power and energy of the different model configurations. Table 4 summarizes all Bitlet model parameters. Table 5 lists all nine Bitlet equations.

PIM performance is captured by six parameters: OC, PAC, CC, $MATs$, R and CT. Note that OC and PAC are just auxiliary parameters used to compute CC. CPU performance is captured by two parameters: BW and DIO. PIM and CPU energy are captured by the $Ebit_{PIM}$ and the $Ebit_{CPU}$ parameters. For conceptual clarity and to aid our analysis, we designate three parameter types: *technological*, *architectural*, and *algorithmic*. Typical values or ranges for the different parameters are also listed in Table 4.

The nine Bitlet model equations determine the PIM, CPU and the combined performance (TP_{PIM}, TP_{CPU}, $TP_{Combined}$), power (P_{PIM}, P_{CPU}, $P_{Combined}$), and energy per computation (EPC_{PIM}, EPC_{CPU}, $EPC_{Combined}$).



	B	C	D	E	F	G	I	L	O	Q	R	S	T	U	V	W
1				1a	1b	1c	1d	1e	1f		2	3a	3b	3c	3d	4
2				Compaction 48bit->16bit							Shifted	1% Filter	1% Filter	1% Filter	1% Filter	Reduction
3					pim cpu		PIM cpu	pim CPU	PIM CPU		Vec-Add	pim cpu	PIM cpu	pim CPU	PIM CPU	PIM cpu
4	Parameter Name	Notation	Units	16b-OR	16b-ADD	16b-MULT.	16b-ADD	16b-ADD	16b-ADD		16b-ADD	32b-CMP	32b-CMP	32b-CMP	32b-CMP	16b-ADD
5	PIM operation complexity	OC	cycles	**32**	**144**	**1,600**	144	144	144		144	**320**	320	320	320	**1,440**
6	PIM Place & Align	PAC	cycles	-	-	-	-	-	-		**512**	-	-	-	-	**1,183**
7	PIM computational complexity	CC=OC+PAC	cycles	32	144	1,600	144	144	144		**656**	**320**	320	320	320	**2,623**
8	PIM cycle time	CT	sec	1.0E-08	1.0E-08	1.0E-08	1.0E-08	1.0E-08	1.0E-08		1.0E-08	1.0E-08	1.0E-08	1.0E-08	1.0E-08	1.0E-08
9	PIM array dimension (Rows)	R	#	1,024	1,024	1,024	1,024	1,024	1,024		1,024	1,024	1,024	1,024	1,024	1,024
10	PIM array count	MATs	#	1,024	1,024	1,024	**16,384**	1,024	**16,384**		1,024	1,024	**16,384**	1,024	**16,384**	**16,384**
11	PIM energy for op (OC=1) per bit	$Ebit_{PIM}$	Joules	1.0E-13	1.0E-13	1.0E-13	1.0E-13	1.0E-13	1.0E-13		1.0E-13	1.0E-13	1.0E-13	1.0E-13	1.0E-13	1.0E-13
12	CPU memory bandwidth	BW	bps	1.0E+12	1.0E+12	1.0E+12	1.0E+12	**1.6E+13**	**1.6E+13**		1.0E+12	1.0E+12	1.0E+12	**1.6E+13**	**1.6E+13**	1.0E+12
13	CPU data in-out bits (CPU Pure)	DIO_{CPU}	bits	**48**	48	48	48	48	48		48	**200**	200	200	200	16
14	CPU data in-out bits (PIM+CPU)	$DIO_{Combined}$	bits	**16**	16	16	16	16	16		16	**3.00**	3.00	3.00	3.00	**0.016**
15	CPU energy per bit transfer	$Ebit_{CPU}$	Joules	1.5E-11	1.5E-11	1.5E-11	1.5E-11	1.5E-11	1.5E-11		1.5E-11	1.5E-11	1.5E-11	1.5E-11	1.5E-11	1.5E-11
16																
17	Entity	Notation	Units													
18	PIM Throughput	TP_{PIM}	GOPS	**3,277**	**728**	**65.5**	**11,651**	**728**	**11,651**		**160**	**328**	**5,243**	**328**	**5,243**	**640**
19	CPU Throughput (CPU Pure)	$TP_{CPU\ (pure)}$	GOPS	**20.8**	**20.8**	**20.8**	**20.8**	**333.3**	**333.3**		**20.8**	**5.0**	**5.0**	**80.0**	**80.0**	**62.5**
20	CPU Throughput (PIM+CPU)	$TP_{CPU\ (Combined)}$	GOPS	62.5	62.5	62.5	62.5	1,000.0	1,000.0		62.5	333.3	333.3	5,333.3	5,333.3	64,000
21	Combined Throughput	$TP_{Combined}$	GOPS	**61.3**	**57.6**	**32.0**	**62.2**	**421.4**	**921.0**		**44.9**	**165.2**	**313.4**	**308.7**	**2,643.9**	**633.3**
22	PIM Power	P_{PIM}	Watts	**10.5**	**10.5**	**10.5**	**167.8**	**10.5**	**167.8**		**10.5**	**10.5**	**167.8**	**10.5**	**167.8**	**167.8**
23	CPU Power	P_{CPU}	Watts	**15.0**	**15.0**	**15.0**	**15.0**	**240.0**	**240.0**		**15.0**	**15.0**	**15.0**	**240.0**	**240.0**	**15.0**
24	Combined Power	$P_{combined}$	Watts	**14.9**	**14.6**	**12.8**	**15.8**	**107.2**	**234.3**		**13.7**	**12.7**	**24.1**	**23.8**	**203.6**	**166.3**
26	CPU Energy per Computation	EPC_{CPU}	J/GOP	**0.72**	**0.72**	**0.72**	**0.72**	**0.72**	**0.72**		**0.72**	**3.00**	**3.00**	**3.00**	**3.00**	**0.24**
27	Combined Energy per Computation	$EPC_{Combined}$	J/GOP	**0.24**	**0.25**	**0.40**	**0.25**	**0.25**	**0.25**		**0.31**	**0.08**	**0.08**	**0.08**	**0.08**	**0.26**

Fig. 5. Throughput and Power comparison of CPU Pure vs. combined PIM+CPU system.

6.2 Applying The Bitlet Model

The core Bitlet model is implemented as a straightforward Excel spreadsheet[7]. All parameters are inserted by the user and the equations are automatically computed. Figure 5 is a snapshot of a portion of the Bitlet Excel spreadsheet that reflects several selected configurations.

Few general notes:

- The spreadsheet can include many configurations, one per column, simultaneously, allowing wide view of potential options to ease comparison.
- For convenience, in each column, the model computes the three related PIM Pure (PIM), CPU Pure (CPU) and the Combined configurations. To support this, the two DIO parameters are needed; one, DIO_{CPU}, for the CPU Pure system, and one (usually lower), $DIO_{Combined}$, for the combined PIM+CPU system. See rows 13-14 in the spreadsheet.
- Determining the OC, PAC and DIO parameters needs special attention. Sections 3.2 and 4.2 detail how to determine these parameters.
- Fonts and background are colored based on the system they represent: blue for PIM, green for CPU and red for combined PIM+CPU system.
- Bold parameter cells with a light background mark items highlighted in the following discussions and are not inherent to the model.

Following is an in-depth dive into the various selected configurations.

Compaction. Cases 1a-1f (columns E-O) describe simple parallel aligned operations. In all these cases, the PIM performs a 16-bit binary computation in order to reduce data transfer between the memory and the CPU from 48 bits to 16 bits. The various cases differ in the operation type (OR/ADD/MULTIPLY, columns E-G), the PIM array count

[7] The spreadsheet will be available in our github repository.

(1024/16384 MATs), and the CPU memory bandwidth (1000/16000 Gpbs) see cases 1b, 1d-1f, rows 4, 10 and 12. Note that in row 3, *"pim"* means a small PIM system (1024 MATs) while *"PIM"* mean a large PIM system (16384 MATs). Same holds for *"cpu"* (1Tbs) and *"CPU"* (16Tbs). In each configuration, we are primarily interested in the difference between the CPU and the combined PIM+CPU system results. Several observations:

- A lower *OC* (row 5) yields higher PIM throughput and combined PIM+CPU system throughput. The combined PIM+CPU system provides significant benefit over CPU for OR and ADD operations, and almost no benefit for MULTIPLY.
- When the combined throughput is close to the throughput of one of its components, increasing the other component has limited value (*e.g.*, in case 1d, using more MATs (beyond 1024) has almost no impact on the combined throughput (61 GOPS), as the maximum possible throughput with the current bandwidth (1000 Gbps) is 62 GOPS).
- When the throughput goes up, so does the power. Using more MATs or higher bandwidth may require higher power than the system TDP. Power consumption of over 200 Watts is likely too high. Such a system has to be slowed down by activating less MATs, enforcing idle time, etc...
- A comparison of the PIM throughput and the CPU throughput (row 18 and 20) provides a hint as to how to speed up the system. Looking at case 1b (column F), the PIM throughput is 728 GOPS while the CPU throughput is 63 GOPS. In this case, it makes more sense to increase the CPU throughput, and indeed, case 1e (row 21, column L), which increases the CPU bandwidth, improves the throughput more than case 1d (column I) does.

Shifted Vector-add. Case 2 (column R) summarizes the example that was widely used in Sections 4.1 , 4.3 and 5.3.

Filter. Case 3a-3d (columns S-V) repeats the example in Section 4.2. It describes a filter that eventually selects 1% of the records and passes a bit-vector to identify the selected items. Each record is of $s = 200$ bits. As in the compaction case above, the four configurations differ in their memory array size $MATs$ and the memory BW. Similar to the compaction case, we can get an idea of how to speed up the system by looking at rows 18 and 20. In this case, the PIM throughput is lower and it makes sense to add more MATs and not memory BW. Indeed, case 3b (column T) with stronger PIM, exhibits higher throughput than case 3c (column U) with higher memory BW.

Reduction. Case 4 (column W) reflects summing all elements in a 16-bit vector. For simplicity, we use the per-MAT reduction method, $Reduction_1$, where all initial per-MAT results are transferred to the CPU. On the CPU side, this computation is like a filter where only one element per MAT ($p = 1/R$) is transferred, and there is no need to transfer a bit-vector. On the PIM side, CC is determined as described in Table 2. With $R = 1024$ rows, the number of phases is $ph = \lceil \log_2(1024) \rceil = 10$. Overall, the CC of the reduction is relatively high, therefore, a PIM-based reduction solution requires many MATs to be more beneficial than a Pure CPU solution.

6.3 Impact and Interplay among Model Parameters

In the previous sub-section, we showed how to determine the throughput and the power of a given system configuration. Now we want to illustrate the sensitivity of the throughput and power to changes in different parameters. In this discussion, due to limited space and limited ability to visualize many parameters concurrently, we focus on the algorithmic and the architectural parameters only, *i.e.*, CC and $MATs$ on the PIM side and DIO and BW on the CPU side. The model itself, as illustrated in Figure 5, supports manipulation of all parameters.

First, in Figure 6, we present the PIM, the CPU and the combined PIM+CPU throughput and power as a function of CC and DIO for a certain PIM and CPU configuration, where $MATs = 1024$ and $BW = 1000$ Gbps. The color on the

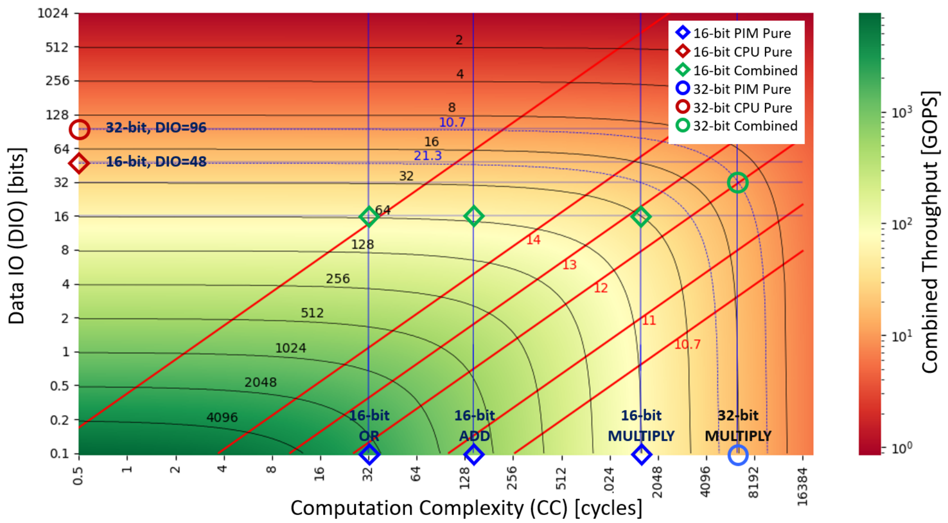


Fig. 6. Combined Throughput [GOPS] and Power [Watt] as function of *CC* and *DIO*. Black curved lines are equal throughput lines. Red curved lines are equal power lines. Blue horizontal (vertical) lines are equal *DIO* (*CC*) lines to allow comparison between PIM, CPU and Combined throughput. Diamond and circle marks legend appears on the top right corner of the graph. Fixed Parameters: $MATs = 1024$ arrays, $R = 1024$ rows, $BW = 1000$ Gpbs, $CT = 10$ns.

graph at point (x, y) indicates the combined PIM+CPU throughput value when PIM $CC = x$ and CPU $DIO = y$. The black curved lines on the graph are equal throughput lines and are annotated with the throughput value in Gpbs. Points below a line have higher throughput and vice-versa. The throughput value at the point ($CC = x$, $DIO = 0$) reflects the PIM Pure throughput when $CC = x$. The throughput value at the point ($CC = 0$, $DIO = y$) reflects the CPU Pure throughput when $DIO = y$. Note that since the axes are in logarithmic scales, the points where $CC = 0$ or $DIO = 0$ do not appear on the graph but can be approximated by looking at the value of the equal throughput line that is close to it. For example, the PIM Pure throughput for $CC = 200$ is approximately 512 GOPS. Blue horizontal (vertical) lines are equal *DIO* (*CC*) lines to allow comparison between PIM, CPU and Combined throughput. Red curved lines on the graph are equal power lines; in this graph, power goes up when going from bottom right to top left. Several points where highlighted on the graph. Diamond-shaped points represent the three 16-bit operations (OR/ADD/MULTIPLY) mentioned in the previous sub-section (cases 1a-1c in Figure 5). The circle-shaped points represent the 32-bit MULTIPLY operation. The relevant operation is marked above the relevant diamond on the X axis. Observing marks with the same shape and operation type allows throughput comparison of the same operation between all three PIM, CPU and combined PIM+CPU configurations. Observations:

- For the same *DIO*, higher *CC* implies lower throughput.
  With higher *CC*, PIM benefits decline, *e.g.*, PIM 32/64 bit MULTIPLY has same/lower throughput than CPU.
- For the same *CC*, higher *DIO* implies lower throughput.
- An equal throughput (black) line has a knee. On the left of the knee, the throughput is impacted mostly by *DIO*, that is, the CPU is the bottleneck in this region. Below the knee, the throughput is impacted mostly by *CC*, *i.e.*, the PIM is the bottleneck in this region.
- For a given case, it is worth looking at three points:
  (1) $x = CC,\ y = 0$, representing the PIM Pure throughput,

(2) $x = 0,\ y = DIO_{CPU}$, representing the CPU Pure throughput,

(3) $x = CC,\ Y = DIO_{PIM}$, representing the combined PIM+CPU system throughput.

- The equal power (red) lines reveal three power regions. The top left reflects the CPU bottle-necked region, where the power is very close to CPU Pure power. The bottom right reflects the PIM bottle-necked region, where the power is very close to PIM Pure power. The power changes mainly around the knee, where each small move to the left changes the power closer to the CPU Pure power and, similarly, each small move down changes the power closer the the PIM Pure power. In the current configuration ($MATs = 1024$, $BW = 1000$ Gpbs), where CPU Pure power is higher than PIM Pure power, left means higher power and down means lower power. Different configurations may exhibit different behaviors.
- The linear behaviour of the power lines reflects the fact that the combined PIM+CPU system power is a linear combination of the PIM Pure and CPU Pure power, where each is weighted according to the share of time they are active. When we multiply CC and DIO by the same number, the time ration remains the same, and so does the combined power.

Table 6 lists the marked points in the graph. The 64-bit MULTIPLY was added to the table to highlight a high computation complexity case where CPU Pure performs better than the combined PIM+CPU configuration.

Table 6. *Throughput of Binary-Operations Examples*

Operation	**16-bit OR**	**16-bit ADD**	**16-bit MULTIPLY**	**32-bit MULTIPLY**	**64-bit MULTIPLY**
CC [cycles]	32	144	1600	6400	25600
DIO CPU / Combined [bits]	48 / 16	48 / 16	48 / 16	96 / 32	192 / 64
PIM Throughput [GOPS]	3277	728	65.5	16.4	4.1
CPU Throughput [GOPS]	20.8	20.8	20.8	10.4	5.2
Combined Throughput [GOPS]	61.3	57.6	32.0	10.7	3.2
PIM Power [Watts]	10.5				
CPU Power [Watts]	15.0				
Combined Power [Watts]	14.9	14.6	12.8	12	11.4

Finally, Figure 7 presents the impact of $MATs$ and BW on throughput and power. This figure assumes a certain pre-defined CC=6400 bits, $DIO_{Combined}$=16 bits, and DIO_{CPU}=48bits. The color on the graph at point (x, y) indicates the combined PIM+CPU throughput value when PIM $MATs = x$ and CPU $BW = y$. The figure has curved and horizontal equal throughput and power lines. Black curved equal throughput lines and magenta curved equal power lines reflect the combined PIM+CPU configuration using DIO=16 bits. Blue horizontal lines reflect the CPU Pure throughput, Magenta horizontal lines reflect the CPU Pure power, both at DIO_{CPU}=48 bits. The diamond marks indicate throughput and power crossover points between natural trade-offs of Pure CPU at DIO=48 bits and combined PIM+CPU at DIO=16 bits. Observations:

- Both throughput and power increase linearly when either $MATs$ or BW increase. The crossover points help compare the CPU Pure and combined PIM+CPU alternatives. When BW is high and $MATs$ is low, the PIM becomes the bottleneck and using CPU Pure is more beneficial than using PIM. On the other hand, when BW is low and $MATs$ is high, the combined PIM+CPU configuration is better.
- The choice of working points out of all points on the same equal throughput or power line depends on the available technology and possible configurations. Bandwidth, memory size, or power limitation leaves only

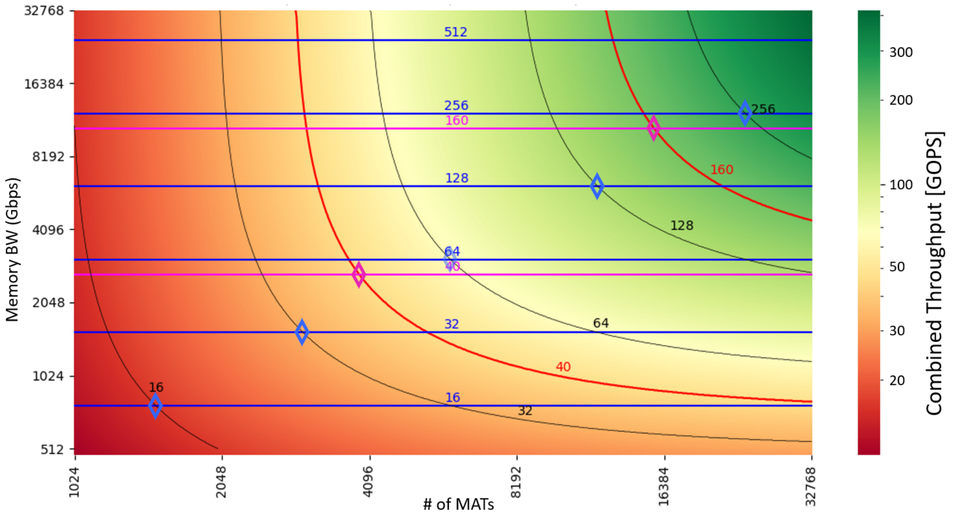


Fig. 7. Combined throughput [GOPS] and power [Watt] as function of number of $MATs$ and memory BW. Black curved lines are equal combined throughput lines. Red curved lines are equal combined power lines, both at $DIO = 16$ bits. Blue/magenta horizontal lines are equal CPU throughput/power lines at $DIO = 48$ bits. Diamond marks indicate crossover point where CPU throughput (power) equals combined throughput (power). Fixed parameters: $CC = 6400$ cycles, $R = 1024$ rows, $DIO = 16/48$ bits, $CT = 10$ns.

part of the space available, *e.g.*, if BW is limited to 4000 Gpbs and memory size is limited to 8192K $MATs$, then roughly only points from the bottom left quarter of the graph are valid. If we also limit the power to 40 Watts, another part of the space becomes invalid.

- We can model a PIM Pure system, by adding vertical lines to reflect PIM Pure power and performance (lines are not shown).

The above are just several examples of the types of analyses that are enabled by the Bitlet model. The model enables analytic exploration of many parameter combinations of PIM and CPU systems.

6.4 Analysis of Real-Life Examples using the Bitlet Model

In the previous sections, we analyzed relatively simple and synthetic examples. In this subsection, we apply the Bitlet model on several real-life examples taken from two PIM-related papers, the Fixed Point Dot Product (FiPDP), the Hadamard Product, and the Image Convolution from the IMAGING paper [14] and the Floating Point multiplication and addition from the FloatPIM paper [19]. All examples map useful and important algorithms onto a MAGIC-based PIM system.

In all examples, the authors made an admirable effort to compute the latency and tried to assess the throughput and power of these computations. In most cases, the authors used a single configuration ($MATs$, R), assumed a single value for the technological parameters, and deduced the throughput, power and energy based on the single values.

The Bitlet model complements the above works nicely. Using their values for CC, the model can easily illustrate the throughput, power, and energy for different parameters. The model can help compare the results to the CPU Pure and the combined PIM+CPU systems.

6.4.1 **IMAGING.** The paper implements several algorithms and analyzes them, but does not consider technological parameters, *e.g.*, cycle time and energy, in the analysis. As a consequence, it presents results in throughput per cycles and determines area based on the number of memory cells.

- **Fixed Point Dot Product (FiPDP)**[8]**.** is a classical dot-product $S = \sum_{i=1}^{N} A_i \times B_i$, where two vectors are multiplied element-wise, and the result vector is summed. Assume two 8-bit vectors producing 16-bit interim results that are summed into 32-bit numbers. The paper assumes $R = 512$. The CC consists of the multiplication step ($12.5 \times 8^2 = 800$ cycles) followed by the reduction step ($ph \times (OC + W) + R - 1 = 9 \times (288 + 32) + 511 \approx 3391$ cycles). The two steps together take approximately 4200 cycles. The paper neither states the throughput of this operation, nor does it make any sensitivity analysis. Using the Bitlet model, we can easily compute the throughput and analyze its sensitivity. For example, for $MATs = 512$, $R = 512$, and $CT = 10ns$, we achieve PIM Pure and combined PIM+CPU throughput of about 6 GOPS, which is rather low compared to the CPU Pure throughput of 31 GOPs at $BW = 1000$ Gpbs. Using a configuration of $MATs = 4096$ and $R = 1024$ increases the PIM Pure (and combined PIM+CPU) throughput to about 100 GOPS, which is higher than the CPU Pure throughput of 31 GOPS stated above.
- **Hadamard Product**[9]**.** The *Hadamard Product* is an element-wise $K \times K$ matrix product, that is, for all (i, j), $C_{i,j} = A_{i,j} \times B_{i,j}$. In fact, it is equivalent to an element-wise N vector product, that is, for all i, $C_i = A_i \times B_i$. The paper focuses on 8-bit pixels as the elements to multiply. If memory space is scarce, several pairs of elements are located in the same row to fit the matrices in the available memory. The paper also considers the case where the input vectors are larger than the size of available memory, so the computation needs to be repeated. None of these manipulations affect the computation throughput since they basically result in doing, *e.g.*, 10× more work in 10× longer time. For throughput computation, we use the Bitlet model assuming a single multiplication in each row. Doing so, we can obtain the real throughput in GOPS and compare it to the CPU Pure and the combined PIM+CPU system throughput.

  Table 7 provides several examples with varying $MATs$ and R values. We assume the paper's original value of $CC = 710$ cycles. We use the Bitlet model to also compute the PIM Pure, CPU Pure and combined PIM+CPU system throughput. For the CPU configuration, we assume $BW = 1000$ Gbps, $DIO = 32$ bits for CPU Pure and $DIO = 16$ bits for the combined system. As expected, the throughput goes up with the number of $MATs$ (and R). For low numbers of $MATs$ and R, CPU Pure is better than a combined PIM+CPU system (31 GOPS vs. 23 GOPS). However, adding MATs improves the combined PIM+CPU system throughput compared to CPU Pure, providing over 49 GOPS vs. 31 GOPS, respectively.

Table 7. *Throughput of the Hadamard Product*

$MATs$	R	CC [Cycles]	$TP_{/cycle}$ [GOP/Cycle]	TP_{PIM} [GOPS]	TP_{CPU} [GOPS]	$TP_{Combined}$ [GOPS]
512	512	710	369	37	31	23
1,024	512	710	738	74	31	34
4,096	1,024	710	5,907	591	31	57
16,384	1,024	710	23,630	2,363	31	61

[8] https://en.wikipedia.org/wiki/Dot_product

[9] https://en.wikipedia.org/wiki/Hadamard_product_(matrices)

- **Image Convolution**[10]**.** A single convolution computation consists of multiplying a $P \times P$ pixel window in a picture with a $P \times P$ coefficient matrix, and creating a new picture where the center element of the selected window is set with the new computed value. P is usually a small odd number, *e.g.*, 3 or 5. Each pixel is W-bits wide. The IMAGING paper goes a long (and smart) way to implement the convolution on top of MAGIC NOR memory array and computing its latency. For our discussion, we need to consider only the following: (a) Computing each pixel involves P^2 W-bit multiplications, $(P^2 - 1)$ $2W$-bit additions, $W \times P \times (P - 1)$ HCOPY operations and $(P - 1)$ VCOPY operations. The last $\frac{P-1}{2}$ pixels in each row (*e.g.*, 1 or 2 pixels for $P = 3,\ 5$) are duplicated at the beginning of the next row. Therefore, to reduce space overhead, each row has to have minimal number of pixels. In the aforementioned examples, $8 + 1 = 9$ pixels per row for $P = 3$ and $8 + 2 = 10$ for $P = 5$. Table 8 lists the CC for convolutions using $W = 8$ bits, $P = 3,\ 5$ and $R = 512,\ 1024$. The table clearly shows that convolution has a very high computation complexity.

Table 8. *Convolution Computation Complexity.*

P	CC ($R = 512$) [Cycles]	CC ($R = 1024$) [Cycles]
3	69,296	77,488
5	188,592	204,976

At this point we can use the Bitlet model and obtain the throughput values. One immediate observation is that since the input and output matrices have the same size, there is no data transfer reduction, and the value of using PIM as a pre-processing stage is questionable. In other words, both DIO_{CPU}=16 and $DIO_{Combined}$=16, thus the CPU Pure throughput is higher than that of the combined PIM+CPU throughput. We ignore this concern and compare PIM Pure to CPU Pure. Results are shown in Table 9. The table shows that convolution is significantly heavier than the previous examples examined above. This is expected, as convolution involves many multiplications per pixel, especially when $P = 5$. According to the model, only a huge PIM configuration ($MATs = 64K, R = 1024$) may compete with CPU Pure. One may question even that, since the power needed for this configuration, obtained from the Bitlet model as well, is approximately 650 Watts. It is worth noting that this computation is quite heavy for the CPU core as well. Every pixel requires (for $P = 3$) 3^2=9 8-bit multiplications and 3^2-1=8 16-bit additions, so to sustain $63G$ convolutions per second, the CPU needs to perform about $63G \times 17 \simeq 1T$ instructions per second. Achieving that throughput requires, for example, four 4-GHz high end CPUs, supporting two wide SIMD instructions (*e.g.*, *AVX-512*[11]) per cycle.

6.4.2 **FloatPIM.** The FloatPIM paper implements a fully-digital scalable PIM architecture that natively supports floating-point operations. As opposed to the IMAGING paper, FloatPIM does address time and power evaluation. In this section, we discuss the in-memory floating point operation used in FloatPIM.

A floating-point multiply operation takes $T_{Mul} = 12N_e + 6.5N_m^2 + 7.5N_m - 2$ cycles, where N_m and N_e are the number of mantissa and exponent bits, respectively. Similarly, floating-point add operation takes $T_{Add} = (3 + 16N_e + 19N_m + N_m^2)$ NOR cycles and $(+2N_m + 1)$ *search* cycles. For simplicity, we assume here that NOR and search cycles have the same

[10] https://en.wikipedia.org/wiki/Kernel_(image_processing)
[11] https://en.wikipedia.org/wiki/AVX-512

Table 9. *Convolution Throughput.*

P	$MATs$	R	CC [Cycles]	$TP_{/cycle}$ [GOP/Cycle]	TP_{PIM} [GOPS]	TP_{CPU} [GOPS]	$TP_{Combined}$ [GOPS]
3	1,024	1,024	77,488	14	1.4	63	1.3
3	8,192	1,024	77,488	108	10.8	63	9.2
3	65,536	1,024	77,488	866	86.6	63	36.3
5	1,024	1,024	204,976	5	0.5	63	0.5
5	8,192	1,024	204,976	41	4.1	63	3.8
5	65,536	1,024	204,976	327	32.7	63	21.5

cycle time. The paper uses the *bfloat16*[12] number format where $N_m = 7$ and $N_e = 8$. Following that, $T_{Mul} = 360$ cycles and $T_{Add} = 328$ cycles. On average, each of the two *bfloat16* operations takes $CC \simeq 344$ cycles.

We tried to approximate the FloatPIM floating-point throughput and power using the Bitlet model. In particular, we tried to understand the sensitivity of these numbers to the technological cycle time and energy model parameters. Bitlet default parameters for CT and $Ebit_{PIM}$ are 10ns and 0.1pJ, respectively. The FloatPIM equivalents are 1.1ns and 0.29fJ. Table 10 shows the significant impact of the model parameters on the results. The first line in the table uses FloatPIM parameters, while the second line uses the Bitlet model default parameters. The results differ a lot, but once shown, seem quite obvious. A 9× faster cycle time increases the throughput by 9×. Reducing energy per bit by 345× increases computation per Joule by 345×, and, finally, accounting the two differences combined, increases power by $\frac{345}{9.1} = 38\times$. Note that FloatPIM uses near-memory functions in addition to in-memory functions to implement *bfloat16* add. Our comparison focuses on highlighting the impact of the model parameter setting, so we have accounted MAGIC-NOR cycles only and ignored the near-memory work.

Table 10. *FloatPIM parameters vs. Bitlet Defaults.*

Model	$MATs$	R	CC [Cycles]	CT [sec]	$Ebit_{PIM}$ [Joule]	$TP_{/cycle}$ [GOP/Cycle]	TP_{PIM} [GOPS]	P_{PIM} [Watt]	TP_{PIM}/P_{PIM} [GOPS/Watt]
FloatPIM	65,536	1024	336.5	1.10E-09	2.90E-16	199,432	181,302	18	10247
Default	65,536	1024	336.5	1.00E-08	1.00E-13	199,432	19,943	671	30

Two observations from the FloatPIM analysis:

- The choice of *bfloat16* is quite beneficial. The *bfloat16* add/mul computation complexity is 328/380 cycles, quite reasonable compared to fixed32 add/mul computation complexity of 288/6400 cycles.
- The choice of technological (and other) parameters has huge impact on the results. The 345× difference in GOPS per Watt is quite significant when comparing a PIM system to a CPU system.

6.5 Model Limitations

As in many models, the Bitlet model trades accuracy with simplicity. In this section, we list several model limitations that Bitlet users should be aware of. Some of these limitations will be addressed in future versions of Bitlet. The list below distinguishes between limitations due to lack of refinement and unsupported features.

Potential model refinement:

[12] https://en.wikipedia.org/wiki/Bfloat16_floating-point_format

- **Inter-MAT Copying.** The model ignores inter-MAT copying (see Section 3.2). Some use cases may require many inter-MAT copies and accounting for them will improve the model accuracy. Adding inter-MAT copying is challenging since it requires modeling of the memory internal busses.
- **Impact of Arithmetic Intensity.** The model assumes that in PIM-relevant workloads, the CPU throughput is solely determined by the data transfer throughput (Section 4.2). This assumption is valid for today's data-intensive applications, and it simplifies the Bitlet model tremendously. If, in the future, the memory bus bandwidth increases to the point it is no longer a bottleneck, the CPU core activity will have to be taken into consideration when assessing the CPU throughput.
- **Cell Initialization.** Depending on the PIM logic technology and the specific basic operation in use, an output cell may need to be initialized before it is computed (*e.g.*, to R_{ON} in MAGIC NOR based PIM). The extra initialization cycles can potentially double the PIM execution time and should be considered in the computation complexity.
- **Row Selection.** When computing power, the current model assumes that at every PIM cycle, all cells in the target column consume energy. This assumption may be false if row selection is used. Counting all rows instead of only the participating rows increases the energy estimate and degrades the model accuracy. This may be significant in algorithms that make serial $VCOPY$s, like shifted vector-add and reductions (see Section 3.2).

Potential new features:

- **Pipelined PIM and CPU.** Until now, we assumed that PIM computation and data transfer cannot overlap. We can work around this restriction by splitting the PIM computation into two parts, each of which uses half of the MATs (using separate banks). Doing so, the PIM and data transfer can be pipelined. As a results, if the memory bus is the bottleneck, the throughput (and power) will increase.
- **Endurance and Lifetime.** The current Bitlet model does not support endurance and lifetime considerations or estimates. Since the model does count the CC cycles, it can help count cell writes, and hence, help in assessing endurance impact on lifetime.
- **Non-single-row Based PIM Computations.** Bitlet assumes the single row-based computing principle, where each row contains a separate computation element and, in each cycle, all of the rows may participate in computation concurrently (Section 2.4). In some PIM use cases, a record may span over more than one row to either improve latency or to locate long data elements within a short row. The model can support such cases, assuming the CC and the R parameters are carefully computed to reflect this.

7 CONCLUSIONS

This paper motivates and describes Bitlet, a parameterized analytical model for comparison of PIM and CPU systems in terms of throughput and power. We explained the PIM computation principles, presented several use cases and demonstrated how the model can be used to analyze real-life examples. We showed how to use the model to pinpoint when PIM is beneficial and when it is not, and to understand the related trade-offs and limits. We believe the model provides insights into how stateful logic-based PIM performs.

We analyzed several selected PIM and CPU systems and some insights following this analysis. For example, the effectiveness of a PIM system depends on several parameters, *e.g.*, the degree of parallelism, data reduction potential and power limitations of the architecture. In our analysis, we stabilized several model parameters and performed only partial analysis of the systems, mainly for demonstration of the model abilities and features. Many more systems can be fully explored by the Bitlet model, and we expect more insights will be reached.

In the future, we plan to extend the Bitlet model and refine it to consider inter-mat copying, impact of arithmetic intensity, cell initialization and row selection. Such model refinements will increase the model accuracy. We also plan to add new features, *e.g.*, endurance and lifetime estimation and non-single-row based PIM evaluation. The former will provide deeper inspection, analysis and comparison of PIM systems, and the latter will significantly expand the span of PIM systems that can be analyzed by the Bitlet model.

REFERENCES

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 105–117.

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

[3] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinsky. 2019. SIMPLER MAGIC: Synthesis and Mapping of In-Memory Logic Executed in a Single Row to Improve Throughput. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (Jul. 2019), 1–1. https://doi.org/10.1109/TCAD.2019.2931188

[4] R. Ben Hur, N. Wald, N. Talati, and S. Kvatinsky. 2017. SIMPLE MAGIC: Synthesis and in-memory Mapping of logic execution for memristor-aided logic. *Proceedings of the 36th International Conference on Computer-Aided Design* (Nov. 2017), 225–232. https://doi.org/10.1109/ICCAD.2017.8203782

[5] Debjyoti Bhattacharjee, Rajeswari Devadoss, and Anupam Chattopadhyay. 2017. ReVAMP: ReRAM Based VLIW Architecture for in-Memory Computing. In *Proceedings of the Conference on Design, Automation Test in Europe* (Lausanne, Switzerland) *(DATE '17)*. European Design and Automation Association, Leuven, BEL, 782–787.

[6] D. Bhattacharjee, Y. Tavva, A. Easwaran, and A. Chattopadhyay. 2020. Crossbar-Constrained Technology Mapping for ReRAM Based In-Memory Computing. *IEEE Trans. Comput.* 69, 5 (2020), 734–748.

[7] Julien Borghetti, Gregory Snider, Philip Kuekes, Jianhua Joshua Yang, Duncan Stewart, and Stan Williams. 2010. Memristive switches enable stateful logic operations via material implication. *Nature* 464 (04 2010), 873–6. https://doi.org/10.1038/nature08940

[8] Julien Borghetti, Gregory S Snider, Philip J Kuekes, J Joshua Yang, Duncan R Stewart, and R Stanley Williams. 2010. 'Memristive' switches enable 'stateful' logic operations via material implication. *Nature* 464, 7290 (Apr. 2010), 873—876. https://doi.org/10.1038/nature08940

[9] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie. 2020. Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey. *Proc. IEEE* (Mar. 2020), 1–48.

[10] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi R. Iyer, Dennis Sylvester, David T. Blaauw, and Reetuparna Das. 2018. Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. *CoRR* abs/1805.03718 (2018). arXiv:1805.03718 http://arxiv.org/abs/1805.03718

[11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. 2012. Dark Silicon and the End of Multicore Scaling. *IEEE Micro* 32, 3 (2012), 122–134.

[12] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2018. In-Memory Data Parallel Processor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3173162.3173171

[13] John L. Gustafson. 1988. Reevaluating Amdahl's Law. *Commun. ACM* 31, 5 (May 1988), 532–533. https://doi.org/10.1145/42411.42415

[14] A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, and S. Kvatinsky. 2018. IMAGING: In-Memory AlGorithms for Image processiNG. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65, 12 (2018), 4258–4271.

[15] Mark Hill and Vijay Janapa Reddi. 2019. Gables: A Roofline Model for Mobile SoCs. 317–330. https://doi.org/10.1109/HPCA.2019.00047

[16] Mark D. Hill and Michael R. Marty. 2008. Amdahl's Law in the Multicore Era. *Computer* 41, 7 (July 2008), 33–38. https://doi.org/10.1109/MC.2008.209

[17] B. Hoffer, V. Rana, S. Menzel, R. Waser, and S. Kvatinsky. 2020. Experimental Demonstration of Memristor-Aided Logic (MAGIC) Using Valence Change Memory (VCM). *IEEE Transactions on Electron Devices* 67, 8 (2020), 3115–3122.

[18] R. B. Hur and S. Kvatinsky. 2016. Memory Processing Unit for in-memory processing. In *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*. 171–172.

[19] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 802–815. https://doi.org/10.1145/3307650.3322237

[20] Mohsen Imani, Saransh Gupta, and Tajana Rosing. 2017. Ultra-Efficient Processing In-Memory for Data Intensive Applications. In *Proceedings of the 54th Annual Design Automation Conference 2017* (Austin, TX, USA) *(DAC '17)*. Association for Computing Machinery, New York, NY, USA, Article 6, 6 pages. https://doi.org/10.1145/3061639.3062337

[21] Jeremie S. Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. 2018. GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies. *BMC Genomics* 19, S2 (May 2018). https://doi.org/10.1186/s12864-018-4460-0

[22] Kunal Korgaonkar, Ronny Ronen, Anupam Chattopadhyay, and Shahar Kvatinsky. 2019. The Bitlet Model: Defining a Litmus Test for the Bitwise Processing-in-Memory Paradigm. arXiv:1910.10234 [cs.AR]

[23] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. 2014. MAGIC—Memristor-Aided Logic. *IEEE Transactions on Circuits and Systems II: Express Briefs* 61, 11 (2014), 895–899.

[24] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. 2014. Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 10 (2014), 2054–2066.

[25] Mario Lanza, H.-S. Philip Wong, Eric Pop, Daniele Ielmini, Dimitri Strukov, Brian C. Regan, Luca Larcher, Marco A. Villena, J. Joshua Yang, Ludovic Goux, Attilio Belmonte, Yuchao Yang, Francesco M. Puglisi, Jinfeng Kang, Blanka Magyari-Köpe, Eilam Yalon, Anthony Kenyon, Mark Buckwell, Adnan Mehonic, Alexander Shluger, Haitong Li, Tuo-Hung Hou, Boris Hudec, Deji Akinwande, Ruijing Ge, Stefano Ambrogio, Juan B. Roldan, Enrique Miranda, Jordi Suñe, Kin Leong Pey, Xing Wu, Nagarajan Raghavan, Ernest Wu, Wei D. Lu, Gabriele Navarro, Weidong Zhang, Huaqiang Wu, Runwei Li, Alexander Holleitner, Ursula Wurstbauer, Max C. Lemme, Ming Liu, Shibing Long, Qi Liu, Hangbing Lv, Andrea Padovani, Paolo Pavan, Ilia Valov, Xu Jing, Tingting Han, Kaichen Zhu, Shaochuan Chen, Fei Hui, and Yuanyuan Shi. 2019. Recommended Methods to Study Resistive Switching Devices. *Advanced Electronic Materials* 5, 1 (2019), 1800143. https://doi.org/10.1002/aelm.201800143 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/aelm.201800143

[26] E Linn, R Rosezin, S Tappertzhofen, U Böttger, and R Waser. 2012. Beyond von Neumann—logic operations in passive crossbar arrays alongside memory operations. *Nanotechnology* 23, 30 (jul 2012), 305205. https://doi.org/10.1088/0957-4484/23/30/305205

[27] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. 1998. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (Barcelona, Spain) *(ISCA '98)*. IEEE Computer Society, USA, 192–203. https://doi.org/10.1145/279358.279387

[28] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. 2017. Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/3123939.3124545

[29] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. 1997. A case for intelligent RAM. *IEEE Micro* 17, 2 (1997), 34–44.

[30] A. Pedram, S. Richardson, M. Horowitz, S. Galal, and S. Kvatinsky. 2017. Dark Memory and Accelerator-Rich System Optimization in the Dark Silicon Era. *IEEE Design Test* 34, 2 (2017), 39–50.

[31] P. Ranganathan. 2011. From Microprocessors to Nanostores: Rethinking Data-Centric Systems. *Computer* 44, 1 (2011), 39–48.

[32] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. . Chen, R. M. Shelby, M. Salinga, D. Krebs, S. . Chen, H. . Lung, and C. H. Lam. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4.5 (2008), 465–479.

[33] J. Reuben, R. Ben-Hur, N. Wald, N. Talati, A. H. Ali, P. Gaillardon, and S. Kvatinsky. 2017. Memristive logic: A framework for evaluation and comparison. *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)* (Sep. 2017), 1–8. https://doi.org/10.1109/PATMOS.2017.8106959

[34] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-Programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 67–80.

[35] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 273–287. https://doi.org/10.1145/3123939.3124544

[36] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. *ACM/IEEE ISCA* (Jun. 2016), 14–26. https://doi.org/10.1109/ISCA.2016.12

[37] Nishil Talati, Ameer Haj Ali, Ben Hur, Nimrod Wald, Ronny Ronen, Pierre-Emmanuel Gaillardon, and Shahar Kvatinsky. 2018. Practical Challenges in Delivering the Promises of Real Processing-in-Memory Machines. https://doi.org/10.23919/DATE.2018.8342275

[38] V. Tenace, R. G. Rizzo, D. Bhattacharjee, A. Chattopadhyay, and A. Calimera. 2019. SAID: A Supergate-Aided Logic Synthesis Flow for Memristive Crossbars. *DATE* (Mar. 2019), 372–377. https://doi.org/10.23919/DATE.2019.8714939

[39] E. Testa, M. Soeken, O. Zografos, L. Amaru, P. Raghavan, R. Lauwereins, P. Gaillardon, and G. De Micheli. 2016. Inversion optimization in Majority-Inverter Graphs. *NANOARCH* (Jul. 2016), 15–20. https://doi.org/10.1145/2950067.2950072

[40] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[41] H. . P. Wong, H. Lee, S. Yu, Y. Chen, Y. Wu, P. Chen, B. Lee, F. T. Chen, and M. Tsai. 2012. Metal–Oxide RRAM. *Proc. IEEE* 100, 6 (2012), 1951–1970.

[42] Dev Narayan Yadav, Phrangboklang L. Thangkhiew, and Kamalika Datta. 2019. Look-ahead mapping of Boolean functions in memristive crossbar array. *Integration* 64 (Jan. 2019), 152 – 162. https://doi.org/10.1016/j.vlsi.2018.10.001