In-Memory Data Parallel Processor

Daichi Fujiki
University of Michigan
dfujiki@umich.edu

Scott Mahlke
University of Michigan
mahlke@umich.edu

Reetuparna Das
University of Michigan
reetudas@umich.edu

Abstract

Recent developments in Non-Volatile Memories (NVMs) have opened up a new horizon for in-memory computing. Despite the significant performance gain offered by computational NVMs, previous works have relied on manual mapping of specialized kernels to the memory arrays, making it infeasible to execute more general workloads. We combat this problem by proposing a programmable in-memory processor architecture and data-parallel programming framework. The efficiency of the proposed in-memory processor comes from two sources: massive parallelism and reduction in data movement. A compact instruction set provides generalized computation capabilities for the memory array. The proposed programming framework seeks to leverage the underlying parallelism in the hardware by merging the concepts of data-flow and vector processing. To facilitate in-memory programming, we develop a compilation framework that takes a TensorFlow input and generates code for our in-memory processor. Our results demonstrate 7.5× speedup over a multi-core CPU server for a set of applications from Parsec and 763× speedup over a server-class GPU for a set of Rodinia benchmarks.

ACM Reference Format:
Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2018. In-Memory Data Parallel Processor . In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3173162.3173171

1 Introduction

Non-Volatile Memories (NVMs) create oppportunities for advanced in-memory computing. By re-purposing memory structures, certain NVMs have been shown to have in-situ analog computation capabilities. For example, resistive memories (ReRAMs) store the data in the form of resistance of titanium oxides, and by injecting voltage into the word line and sensing the resultant current on the bit-line, the *dot-product* of the input voltages and cell conductances is obtained using Ohm's and Kirchhoff's laws.

Recent works have explored the design space of ReRAM-based accelerators for machine learning algorithms by leveraging this *dot-product* functionality [13, 39]. These ReRAM-based accelerators exploit the massive parallelism and relaxed precision requirements, to provide orders of magnitude improvement when compared to current CPU/GPU architectures and custom ASICs, in-spite of their high read/write latency. *In this paper, we seek to answer the question, to what extent is resistive memory useful for more general-purpose computation?*

Despite the significant performance gain offered by computational NVMs, previous works have relied on manual mapping of convolution kernels to the memory arrays, making it difficult to configure it for diverse applications. We combat this problem by proposing a programmable in-memory processor architecture and programming framework. A general purpose in-memory processor has the potential to improve performance of data-parallel application kernels by an order of magnitude or more.

The efficiency of an in-memory processor comes from two sources. The first is massive data parallelism. NVMs are composed of several thousands of arrays. Each of these arrays are transformed into a single instruction multiple data (SIMD) processing unit that can compute concurrently. The second source is a reduction in data movement, by avoiding shuffling of data between memory and processor cores. Our goal is to design an architecture, establish the programming semantics and execution models, and develop a compiler, to expose the above benefits of ReRAM computing to general purpose data parallel programs.

The in-memory processor architecture consists of memory arrays and several digital components grouped in tiles, and a custom interconnect to facilitate communication between the arrays and instruction supply. Each array acts as a unit of storage as well as a vector processing unit. The proposed architecture extends the ReRAM array to support in-situ operations beyond dot product (i.e., addition, element-wise multiplication, and subtraction). We adopt a SIMD execution model, where every cycle an instruction is multi-casted to a set of arrays in a tile and executed in lock-step. The Instruction Set Architecture (ISA) for in-memory computation

ACM ISBN ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173171

consists of 13 instructions. The key challenge is developing a simple yet powerful ISA and programming framework that can allow diverse data-parallel programs to leverage the underlying massive computational efficiency.

The proposed programming model seeks to utilize the underling parallelism in the hardware by merging the concepts of data-flow and vector processing (or SIMD). Data-flow explicitly exposes the Instruction Level Parallelism (ILP) in the program, while vector processing exposes the Data Level Parallelism (DLP). Google's TensorFlow [1] is a popular programming model for machine learning. We observe that TensorFlow's programming semantics is a perfect marriage of data-flow and vector-processing that can be applied to more general applications. Thus, our proposed programming framework uses TensorFlow as the input.

We develop a *TensorFlow compiler* that generates binary code for our in-memory data-parallel processor. The TensorFlow (TF) programs are essentially Data-Flow Graphs (DFG) where each operator node can have multi-dimensional vectors, or tensors, as operands. A DFG that operates on one element of a vector is referred to as a module by the compiler. The compiler transforms the input DFG into a collection of data-parallel modules with identical machine code. Our execution model is coarse-grain SIMD. At runtime, a code module is instantiated many times and processes independent data elements. The programming model and compiler support restricted communication between modules: reduce, scatter and gather. Our compiler explores several interesting optimizations such as unrolling of high-dimensional tensors, merging of DFG nodes to utilize n-ary ReRAM operations, pipelining compute and write-backs, maximizing ILP within a module using VLIW style scheduling, and minimizing communication between arrays.

For general purpose computation, we need to support a variety of compute operations (e.g., division, exponent, square root). These operations can be directly expressed as nodes in TensorFlow's DFG. Unfortunately, ReRAM arrays cannot support them natively due to their limited analog computation capability. Our compiler performs an instruction lowering step in the code-generation phase to translate higher-level TensorFlow operations to the in-memory compute ISA. We discuss how the compiler can efficiently support complex operations (e.g., division) using techniques such as the Newton-Raphson method which iteratively applies a set of simple instructions (add/multiply) to an initial seed from the look-up table and refines the result. The compiler also transforms other non-arithmetic primitives (e.g., square and convolution) to the native memory SIMD ISA.

In summary, this paper offers the following contributions:

- We design a processor architecture that re-purposes resistive memory to support data-parallel in-memory computation. In the proposed architecture, memory arrays store data and act as vector processing units. We extend the ReRAM memory array to support in-situ operations beyond the dot product and design a simple ISA with limited compute capability.
- We develop a compiler that transforms DFGs in Google's TensorFlow to a set of data-parallel modules and generates module code in the native memory ISA. The compiler implements several optimizations to exploit underlying hardware parallelism and unique features/constraints of ReRAM-based computation.
- Although the in-memory compute ISA is simple and limited in functionality, we demonstrate that with a good programming model and compiler, it is possible to off-load a large fraction of general-purpose computation to memory. For instance, we are able to execute in memory an average of 87% of the PARSEC applications studied.
- Our experimental results show that the proposed architecture can provide overall speedup of 7.5× over a state-of-art multicore CPU for the PARSEC applications evaluated. It also provides a speedup of 763× over state-of-art GPU for the Rodinia kernel benchmarks evaluated. The proposed architecture operates with a thermal design power (TDP) of 415 W, improves the energy efficiency of benchmarks by 230× and reduces the average power by 1.26×.

2 Processor Architecture

We propose an in-memory data-parallel processor on ReRAM substrate. This section discusses the proposed microarchitecture, ISA, and implementation of the ISA.

2.1 Micro-architecture

The proposed in-memory processor adopts a tiled architecture as shown in Figure 1. A tile is composed of clusters of memory nodes, few instruction buffers and a router. Each cluster consists of a few memory arrays, a small register file, and look-up table (LUT). Each memory array is shown in Figure 1 (b). Internally, a memory array in the proposed architecture consists of multiple rows of resistive bit-cells, a set of digital-analog converters (DACs) feeding *both* the word-lines and bit-lines, sample and hold circuit (S+H), shift and adder (S+A) and analog-digital converters (ADCs). The process of reading and writing to ReRAM memory arrays remains unchanged. We refer the reader to ReRAM literature for details [39, 42]. The memory arrays are capable of both data storage and computation. We explain the compute capabilities of the memory arrays and the role of digital components (e.g. register file, S+A, LUT) in Section 2.2.

The tiles are connected by an H-Tree router network. The H-Tree network is chosen to suit communication patterns typical in our programming model (Section 3) and it also provides high-bandwidth communication for external I/O. The clusters inside a tile are connected by a router or a

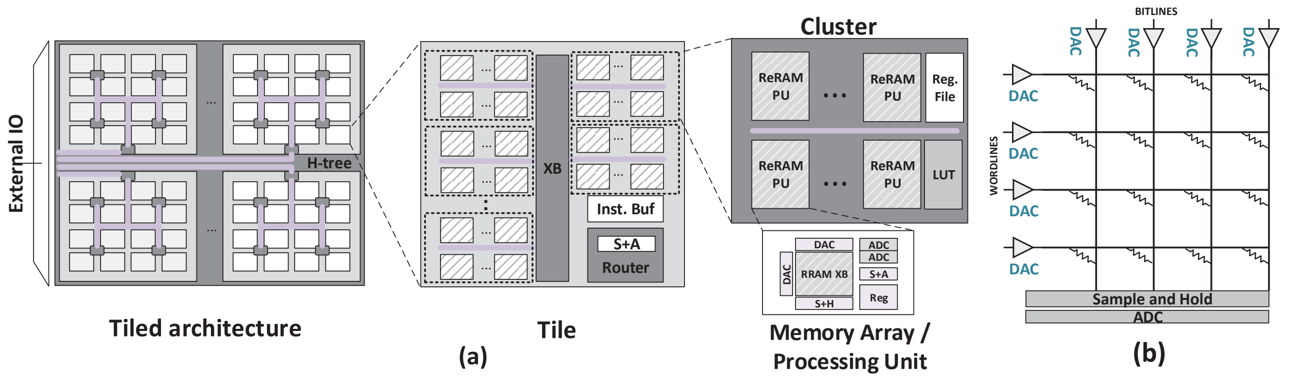

Figure 1. In-Memory Processor Architecture. (a) Hierarchical Tiled Structure (b) ReRAM array Structure

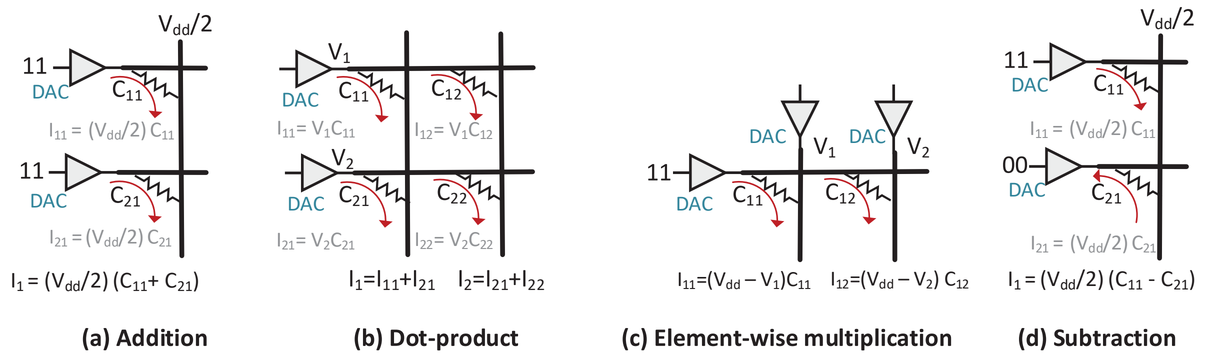

Figure 2. In-situ ReRAM array operations.

crossbar topology. A shared bus facilitates communication inside a cluster. A hierarchical topology inside the tile limits the network power consumption, while providing sufficient bandwidth for infrequent communication typical in data-parallel applications.

Each memory array can be thought of as a vector processing unit with few SIMD lanes. The processor adopts a SIMD execution model. Each array is mapped to a specific instruction buffer. All arrays mapped to the same instruction buffer execute the same instruction. Every cycle, one instruction is read out of the each instruction buffer and multi-casted to the memory arrays in the tile. The execution model is discussed in detail in Section 4.

The processor evaluated in this paper consists of 4,096 tiles, 8 clusters per tile, and 8 memory arrays per cluster. Each array can store 4KB of data and has 8 SIMD lanes of 32 bits each. Consequently, the processor has aggregate SIMD width of two million lanes, aggregate memory capacity of 1GB and 494 mm^2 area. The resolution of ADC and DAC is set to 5 and 2 bits.

2.2 Instruction Set Architecture

The proposed Instruction Set Architecture (ISA) is simple and compact. Compared to a standard SIMD ISA, In-memory ISA does not support complex (e.g. division) and specialized (e.g. shuffle) instructions because these are hard to do in-situ in-memory. Instead, compiler transforms complex instructions to a set of `lut`, `add` and `mul` instructions as discussed later. The ISA consists of 13 instructions as shown in Table 1. Each ReRAM arrays executes the instruction locally, hence the operand addressing modes reference rows inside the array or local registers. The instructions can have a size of up to 34 bytes. Now we discuss the functionality and implementation of individual instructions.

1) add The add instruction is an n-ary operation that adds the data in rows specified by <mask>. The <mask> is a 128-bit mask which is set for each row in the array that participates in addition. Figure 2 (a) shows an add operation. The mask is fed to word-line DACs, which is used to apply a Vdd ('11') or Vdd/2 ('10') to the word-lines. A '1' in the mask activates a row. Each bit-cell in a ReRAM array can be abstractly thought of as variable resistor. Addition is performed inside the array by summing up currents generated by conductance (=resistance^{-1}) of each bit-cell. A sample and hold (S + H) circuit receives the bit-line current and feeds it the ADC unit which outputs the digital value for the current. The result from each bit-line represents the partial sum for bits stored in that bit-line. A word or data element is stored across multiple bit-lines. An external digital shifter and adder (S + A) combines the partial sums from bit-lines. The final result

Opcode	Format	Cycles
add	<mask><dst>	3
dot	<mask><reg_mask><dst>	18
mul	<src><src><dst>	18
sub	<mask><mask><dst>	3
shift{l|r}	<src><dst><imm>	3
mask	<src><dst><imm>	3
mov	<src><dst>	3
movs	<src><dst><mask>	3
movi	<dst><imm>	1
movg	<gaddr><gaddr>	Variable
lut	<src><dst>	4
reduce_sum	<src><gaddr>	Variable

Table 1. In-Memory Compute ISA. The instructions use operand addresses specified by either <src>, <dst> or <gaddr>. The <src> and <dst> is a 8-bit local address (1-bit indicates memory/register + 7-bit row number/register number). The <gaddr> is a 4 byte global address (12-bit tile # + 6-bit array # + 7-bit row # + reserved bits). The <imm> field is a 16 byte immediate value.

is written back to <dst> memory row or register. Each of ReRAM crossbar (XB), ADC and S+A takes 1 cycle, resulting in 3 cycles in total.

2) dot The `dot` instruction is also an n-ary operation which emulates a dot product over the data in rows specified by <mask>. A dot product is a *sum of products*. The *sum* is done using current summation over the bit-line as explained earlier. Each row computes a *product* by streaming in the multiplicand via the word-line DAC in a serial manner as shown in Figure 2 (b). The multiplicands are stored in register file and the individual registers are specified using <reg_mask> field.

Robust current summation over ReRAM bit-lines has been demonstrated in prior works [20, 43]. We adapt the dot product architecture from ISAAC [39] for our `add` and `dot` instructions. We refer the reader to these works for further implementation details.

3) mul The `mul` instruction is 2-ary operation that performs element-wise multiplication over elements stored in the *two* <src> memory rows and stores the result in <dst>. To implement this instruction we utilize the row of DACs at the top of the array feeding the bit-lines (Figure 1 (c)). The multiplicand is streamed in through the DACs serially 2-bits at time and the product is accumulated over bit-lines as shown in Figure 2 (c). The word-line DACs are set to Vdd ('11').

Note that element-wise multiplication was not supported in prior works on memristor-based accelerators, and is a new feature we designed for supporting general purpose data-parallel computation. Since dot product uses the same multiplicand for all elements stored in a row, it can not be utilized for element-by-element multiplication. We solve this problem by using an additional set of DACs for feeding bit-lines. As in ISAAC, the operation is pipelined into 3 stages: XB, ADC and S+A, processing 2 bits per cycle, resulting in 18 cycles in total for 32 bit data.

4) sub The `sub` instruction performs element-wise subtraction over elements stored in the two set of memory rows (minuends and subtrahends) specified by <mask>s and stores the result in <dst>. Subtraction in ReRAM arrays has not been explored before. We support this operation by draining the current via word-line as shown in Figure 2 (d). The output voltage for word-line DAC of the subtrahend row is set to ground allowing for current drain. Hence the remaining current over the bit-line represents the difference between minuend and subtrahend. For this operation we reverse the voltage across memristor bit-cell. Fortunately, several reports on fabricated ReRAM demonstrate the symmetric V/I properties of memristor with reverse voltage across terminals [36, 44].

5) lut The `lut` instruction sends the value stored in <src> as an address to the lookup table (LUT), and write back the data read from the LUT to <dst>. The multi-purpose LUT is implemented for supporting high-level instructions. LUT is utilized for nonlinear functions such as sigmoid, and initial seeding of division and transcendental functions (Section 5.1). The LUT has 512 entries of 8-bit numbers to suffice the precision requirement of the arithmetic algorithms implemented [16]. LUT is a small SRAM structure which operates at much higher frequency than ReRAM arrays and hence shared by multiple arrays. Its contents are initialized by the host at runtime. `lut` takes 4 cycles, adding 1 cycle on top of the basic XB, ADC, S+A pipeline.

6) mov, movi, movg, movs The `mov` family of instructions facilitates movement of data between memory rows of an array, registers, and even across arrays via global addressing (<gaddr>). The global addresses are handled by the network, hence the latency of gobal moves (`movg`) is variable. Immediate values can be stored to <dst> as well via `movi` instruction. These instructions are implemented using traditional memristor read/write operations. The selective mov (`movs`) instruction selectively moves data to elements in <dst> based on an 8-bit mask. Recall that any <dst> row can store 8 32-bit elements in the prototype architecture.

7) reduce_sum The `reduce_sum` instruction sums up the values in the <src> row of different arrays. The reduction is executed outside the arrays. This instruction utilizes the H-tree network and the adders in the routers to reduce values across the tiles.

8) shift / mask The `shift` instruction shifts each of the vector element in <src> by <imm> bits. The mask instruction logically ANDs each of the vector element in <src> with <imm>. These instructions utilize the digital shift and adder (S+A) outside the arrays.

Discussion Our goal is two-fold. First, keep the instruction set as simple as possible to reduce design complexity and retain area efficiency (hence memory density). Second, expose all compute primitives which can be done in-situ inside the memory array without reading the data out. The proposed ISA does not include any instructions for looping, branch or jump instructions. We rely on the compiler to unroll loops wherever necessary. Our SIMD programming model ensures small code size, in spite of unrolling. Control flow is facilitated via condition computation and selective moves (Section 3). The compute instructions in the ISA are restricted to add, sub, dot, mul. Our programming model based on TensorFlow, supports a rich set of compute operations. Our compiler transforms them to a combination of ISA instructions (Section 5.1) and hence enables general purpose computation.

2.3 Precision and Signed Arithmetic

Floating point operations need normalization based on exponent, hence in-memory computation for the floating point operands encumbers huge complexity. We adopt a fixed point representation. We give the flexibility for deciding the position of the decimal point to trade-off between precision and range. But the responsibility to prevent bit overflow and underflow is left to the programmers. We developed a testing tool that can calculate the dynamic range of the input that assures the required precision. Note that under the condition that overflow/underflow does not happen, fixed point representation gives better accuracy compared to floating point. Section 6 discusses the impact on application output.

For general purpose computation, it is important to support negative values. Prior work [39] uses a biased representation for numbers, and then normalizes the bias via subtraction outside the memory arrays. This approach is perhaps reasonable for CNN dot products, because the overhead of subtraction outside the array for normalizing the bias, is compensated by multi-row addition within the array. In general, data-parallel programs' additions need not span multiple rows (often 2 rows are sufficient). In such a scenario, subtraction outside the array needs additional array read which offsets the benefit of biased addition inside the array.

We observe that for b-bit bit-cells (i.e. 2^b resistance levels), current summation followed by shift+adder across bit-lines outputs the correct results as long as negative numbers are stored in 2^b's complement notation. In our prototype design, arrays have 2-bit bit-cell, hence addition over negative numbers stored as 4'complement will yield correct results. Furthermore it can be mathematically proved that 4's complement is exactly equal to 2's complement in base-4 representation. Thus there is no need for conversion between number formats. The same principle holds true for multiplication as long as the DAC used for streaming in the multiplicand has same resolution as resistance level of ReRAM bit-cells. In our design, 2-bit DACs are required.

3 Programming Model

We choose Google's TensorFlow [1] as the programming front-end for proposed in-memory processor. By using TensorFlow, programmers write the kernels which will be offloaded to the memory. TensorFlow expresses the kernel as a *Data Flow Graph* (DFG). Since TensorFlow is available for variety of programming languages (e.g. Python, C++, Java, Go), programmers can easily plug in the TensorFlow kernels in their code. Also, since TensorFlow supports variety of target hardware systems (e.g. CPU, GPU, distributed system), programmers can easily validate the functionality of the kernel and scale the system depending on the input size.

TensorFlow (TF) offers a suitable programming paradigm for data parallel in-memory computing. First, nodes in TF's DFGs can operate on multi-dimensional matrices. This feature embeds the SIMD programming model and facilitates easy exposure of Data Level Parallelism (DLP) to the compiler. Second, irregular memory accesses are restricted by not allowing subscript notation. This feature benefits both programmers and compilers. Programmers do not have to convert high-level data processing operations (e.g., vector addition) into low-level procedural representations (e.g., for-loop with memory access). The compiler can fully understand the memory access pattern. Third, the DFG naturally exposes Instruction Level Parallelism (ILP). This can be directly used by a compiler for Very Long Instruction Word (VLIW) style scheduling to further utilize underlying parallelism in the hardware without implementing complex out-of-order execution support. Finally, TensorFlow supports a persistent memory context in nodes of the DFG. This is useful in our merged memory and compute architecture for storing persistent data across kernel invocations.

Our programming model and compilation framework support the following TensorFlow primitives (See Table 2 for the list of supported TF nodes.):

Input nodes The proposed system supports three kinds of input: Placeholder, Const, and Variable. Placeholder is a non-persistent input and will not be used for future module invocations. Const is used to pass constants whose values are known at compile time. Scalar constants are included in ISA, and vector constants are stored in either the register file or an array based on the type of their consumer node in the DFG. Variable is the input with persistent memory context, of which data can be used and updated in the future kernel invocations. Variables are initialized at kernel launch time.

Operations The framework supports a variety of complex operation nodes including transcendental functions. We discuss the process of lowering these operation nodes into native memory ISA in Section 5.1.

Input nodes	Const	Placeholder	Variable							
Arithmetic Operations	Abs	Add	ArgMin	Div	Exp	FloorDiv	Less	Mul	RealDiv	Sigmoid
	Sqrt	Square	Sub	Sum	Conv2D*	ExpandDims*	MatMul*	Reshape*	Tensordot*	
Control Flow etc.	Assign	AssignAdd	Gather	Identity	Pack	Select	Stack	NoOp		

Table 2. Supported TensorFlow Nodes. (* has restrictions on function/data dimension.)

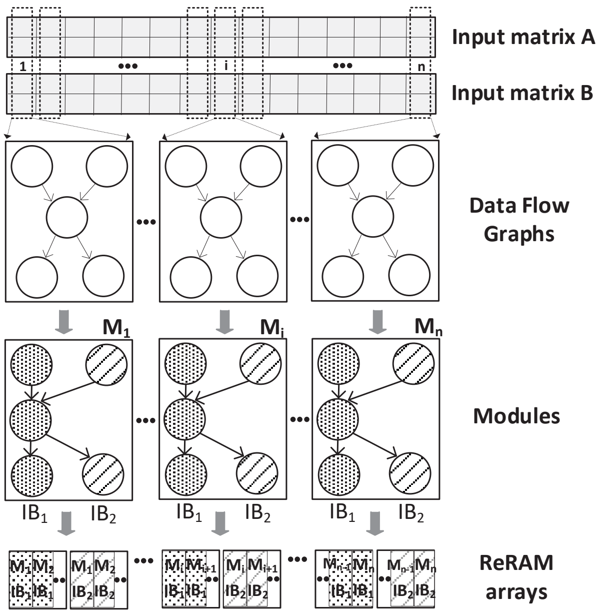

Figure 3. Execution Model.

Control Flow Control flow is supported by a select instruction. A select instruction takes three operands and generates output as follows:

$O[i] = Cond[i] \ ? \ A[i] : B[i].$

A select instruction is converted into multiple selective move (`movs`) instructions. The Condition variable is precomputed and used to generate the mask for the selective moves.

Reduction, Scatter, Gather A reduction node is supported by the compiler and natively in the micro-architecture. Scatter and gather operations are used to implement an indirect reference to the memory address given in the operand. These operations generate irregular memory accesses and require synchronizations to guarantee consistency. Because of the non-negligible overhead, these operations should be used rarely. We observe in many cases that these operations can be eliminated before offloading the kernel by sending gathered data from CPU.

4 Execution Model

The proposed architecture processes data in a SIMD execution model at the granularity of *module*. At runtime, different instances of a module execute the same instructions on different elements of input vectors in a lock-step manner. Our compiler generates a module by unrolling a single dimension of multi-dimensional input vectors as shown in Figure 3. Intuitively, a DFG generated by TensorFlow can represent one module. At kernel launch time, the number of module instances are dynamically created in accordance with the input vector length.

The proposed execution model allows restricted communication between instances of modules. Such communication is only allowed using scatter/gather nodes or reduction nodes in the DFG. We find these communication primitives are sufficient to express most TensorFlow kernels.

Each module is composed of one or more Instruction Blocks (IB) as shown in Figure 3. An IB consists of a list of instructions which will be executed sequentially. Conceptually, an IB is responsible for executing a group of nodes in the DFG. Multiple IBs in a module may execute in parallel to expose ILP. The compiler explores several optimizations to increase the number of concurrent IBs in a module and thereby exposes the ILP inside a module.

We view rows in the ReRAM array as a SIMD vector unit with multiple lanes or *SIMD slots*. Each IB is mapped to a single lane or one slot. To ensure full utilization of all SIMD lanes in the array, the runtime maps identical IBs from different instances of the same module to an individual array as shown in the last row of Figure 3. This mapping results in correct execution because all instances of a module have the same set of IBs. Furthermore, IBs of a module are greedily assigned to nearby arrays so that the communication latency between IBs is minimized.

5 Compiler

The overall compilation flow is shown in Figure 4. Our compiler takes Google's TensorFlow DFG in the protocol buffer format as an input, optimizes it to leverage parallelism that the in-memory architecture offers, and generates executable code for the in-memory processor ISA. The compiler first analyzes the semantics of input DFG which has vector/matrix operands and creates a module with a single IB with required control flow. Several optimizations detailed later expand a module to expose intra-module parallelism by decomposing and replicating the instructions in the single IB into multiple IBs and merging redundant nodes. This is followed by instruction lowering, scheduling of IBs in a module, and code generation. Instruction lowering transforms complex DFG nodes into simpler instructions supported by in-memory processor ISA. Instruction lowering is also done by promoting the specific instructions (e.g. ABS) to general ones (e.g., MASK) and expanding the instruction into a set of native memory ISA instructions.

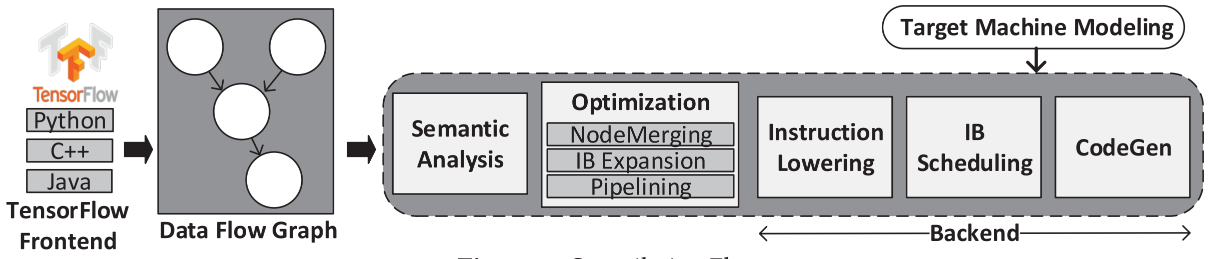

Figure 4. Compilation Flow.

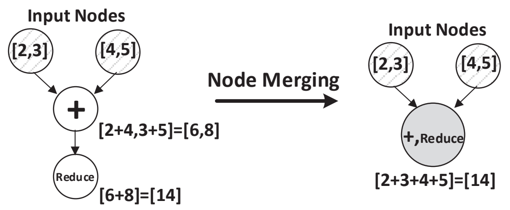

Figure 5. Node Merging.

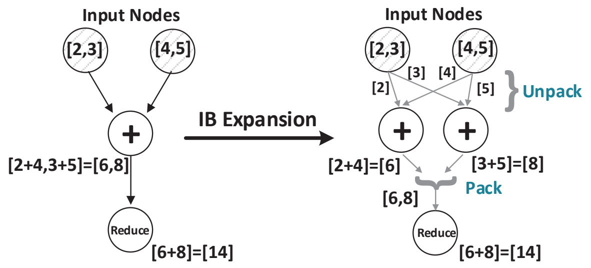

Figure 6. IB Expansion.

The compiler tool-chain is developed using Python 3.6 and C++. The compiler front-end uses TensorFlow's core framework to parse the TensorFlow Graph. TensorFlow nodes supported at this time are listed in Table 2.

5.1 Supporting Complex Operations

The target memory ISA is quite simple and supports limited number of compute instructions as described in Section 2.2. Natively, the arrays can execute dot product, addition, multiplication and subtraction. However, general purpose computation requires supporting a diverse set of operations ranging from division, exponents, transcendental functions, etc. We support these complex operations by converting them into a set of LUT, addition and multiplication instructions based on algorithms used in Intel IA-64 processors [14, 19].

The compiler uses either Newton-Raphson or Maclaurin-Goldschmidt methods that iteratively apply a set of instructions to an initial seed from the look-up table and refine the result. Our implementation chooses the best algorithms based on the precision requirement. We could have used simpler algorithms (e.g., SRT division), but we employ iterative algorithms because (1) bit shift cannot be supported in the array, so for each bit shift operation the values need to be read out and written back, (2) supporting bit-wise logical operations (and, or) are challenging because of multi-level resistive bit-cells, and (3) simple algorithms often require more space, which is challenging for the data carefully aligned in the array.

Finally, the compiler also lowers convolution nodes in the DFG to the native memory ISA. Prior works [39] have mapped convolution filter weights to the array and performed dot product computation by streaming in the input features. Because filters used for general-purpose programs are typically small (e.g. 3x3 for HotSpot and Sobel filter), we map the input data to the array and stream in the filter. This approach reduces buffering for the input data and improves array utilization. Furthermore, the compiler decomposes the convolution into a series of matrix-vector dot-products done simultaneously on different input matrix slices, thereby reducing the convolution time significantly.

5.2 Compiler Optimizations

Node Merging A node merging pass is introduced to fill the gap between the capabilities of the target in-memory architecture and the expressibility of the programming language. The proposed in-memory ISA can support compute operations over n-operands. A node merging pass promotes a series of 2-operand compute nodes in the DFG of a module, to a single compute node with many operands as shown in Figure 5. The maximum number of operands n is limited by the number of array rows and the resolution of ADCs. ADCs consume a significant fraction of chip power, and their power consumption is proportional to their resolution. Our compiler can generate code for an arbitrary resolution n and the chip architects can choose a suitable n based on the power budget.

The node merging pass also combines certain combinations of nodes to reduce intermediate writes to memory arrays. For example, a node which feeds its results to a multiplication node need not write back the results to memory. This is because multiplicand is directly streamed into the array from registers.

Instruction Block Scheduler Independent Instruction Blocks (IBs) inside a module can be co-scheduled to maximize ILP as shown in the third row of Figure 3. Our compiler adapts the Bottom-Up-Greedy (BUG) algorithm [15] for scheduling IBs. BUG was first used in the Bulldog VLIW compiler [15] and has been adapted in various schedulers for VLIW/data-flow architecture, e.g. Multiflow compiler [29] and compiler for the tiled data-flow architecture, WaveScalar [30]. Our implementation of the BUG algorithm first traverses the DFG through a bottom-up path, collecting candidate assignments of the instructions. Once the traversal path reaches the input (define) node, it traverses a top-down path to make a final assignment, minimizing the data transfer latency by taking both the operand location and successor location into consideration. We modify the original BUG algorithm to introduce the notion of in-memory computing, where a functional unit is identical to the data location. We also modified the algorithm to take into account read/write latency, network resource collision latency, and operation latency.

Instruction Block (IB) Expansion Instruction Blocks that use multi-dimensional vectors as operands can be expanded into several instruction blocks with lower-dimension vectors to further exploit ILP and DLP. For example, consider a program that processes 2D matrices of dimension sizes [2, 1024]. The compiler will first convert the program to a module which will be instantiated 1,024 times and executed in parallel. Each module will have an IB that processes 2D vectors. The expansion pass will further decompose the module into 2 IBs that process 1D scalar value.

The expansion pass traverses the nodes in a module's DFG in a bottom-up/breath-first order and detects the subtrees that process multi-dimensional vectors of the same size. The subtree regions detected are expanded. To ensure the dimensions are consistent between the sub-tree regions, pack and unpack pseudo operations are inserted between these regions. Pack and unpack operations are later converted to `mov` instructions. A simplified example is shown in Figure 6.

Pipelining A significant fraction of the compute instructions goes through two phases: compute and write-back. Unfortunately, these two phases are serialized, since an array cannot compute and write simultaneously. Our compiler breaks this bottleneck by pipelining these phases and ensuring the destination address for the write-backs are in a separate array. By using two arrays, one array computes while writing back the previous result to the other array. In the worst case, this optimization lowers the utilization of arrays by half. Thus, this optimization is beneficial when the number of modules needed for the input data is lower than the aggregate SIMD capacity of the memory chip.

Balancing Inter-Module and Intra-Module Parallelism Some of the optimizations discussed above attempt to improve performance by exposing parallelism inside a module. Because of Amdahl's law, increasing the number of IBs in a module will not result in linear speedup. Depending on the data characteristics, the SIMD slots assigned to a module may not be fully utilized in every cycle. In fact, expanding a module could slow down the total execution time when the number of IBs across all module instances exceeds the aggregate SIMD slots in the memory chip. In such a scenario, multiple iterations may be needed to process all module instances, resulting in a performance loss.

Our compiler can generate code for arbitrary upper bounds on the number of IBs per module, and can flexibly tune the intra-module parallelism with respect to inter-module parallelism. We develop a simple analytical model to compute the approximate execution time given the number of IBs per module and number of module instances. The number of module instances is dependent on input data size, and is only known at runtime. Thus, the optimal code is chosen at runtime based on the analytical model and streamed in to the memory chip from host.

	Benchmark	Input data shape	# IB insts.
PARSEC	Blackscholes	[4, 10000000]	163
	Canneal	[2, 600, 4096]	6
	Fluidanimate	[3, 17, 229900]	294
	Streamcluster	[2, 128, 1000000]	6
Rodinia	Backprop	[16, 65536]	117
	Hotspot	[1024, 1024]	26
	Kmeans	[34, 494020]	91
	StreamclusterGPU	[2, 256, 65536]	6

Table 3. Evaluated workloads. Numbers in bracket indicates size of respective x,y,z dimensions

6 Methodology

Benchmarks We use a subset of benchmarks from PARSEC multi-threaded CPU benchmark suite [8] and Rodinia GPU benchmark suite [11] as listed in Table 3. We re-write the kernels of the benchmarks in TensorFlow code and then generate in-memory ISA code using our compiler. We choose to port the applications which could be easily transformed to Structure of Array (SoA) code for the ease of porting to TensorFlow and a data-parallel architecture. We leave the remaining benchmarks to future work. For the benchmarks which use floating point numbers in the kernel, we assess the effect of converting it into fixed point numbers. By tuning the decimal point placement, we ensure that the input data is in the dynamic range of fixed point numbers. We ensure that the quality of result requirement defined by the benchmark is met. We use the native dataset for each benchmark and compare it with the native execution on the CPU and GPU baseline systems. The size of the input for each kernel invocation ranges from 8MB to 2GB.

Area and Power Model All power/area parameters are summarized in Table 4. We use CACTI to model energy and area for registers and LUTs. The energy and area model for ReRAM processing unit, including ReRAM crossbar array,

Component	**Params**	**Spec**	**Power**	**Area(mm^2)**
ADC	resolution frequency number	5 bits 1.2 GSps 64 × 2	64 mW	0.0753
DAC	resolution number	2 bits 64 × 256	0.82 mW	0.0026
S+H	number	64 × 128	0.16 mW	0.00025
ReRAM Array	number	64	19.2 mW	0.0016
S+A	number	64	1.4 mW	0.0015
IR OR Register	size size size	2KB 2KB 3KB	1.09 mW 1.09 mW 1.63 mW	0.0016 0.0016 0.0024
XB	bus width size	16B 10 × 10	1.51 mW	0.0105
LUT	number	8	6.8 mW	0.0056
Inst. Buf	size	8 × 2KB	5.83 mW	0.0129
Router	flit size num_port	16 9	0.82 mW	0.00434
S+A	number	1	0.05 mW	0.000004
1 Tile Total			101 mW	0.12
Inter-Tile Routers	number	584	0.81 W	2.50
Chip total			416 W	494 mm^2

Table 4. In-Memory Processor Parameters

sample-and-hold circuits, shift-and-add circuits are adapted from the ISAAC [39]. We employ energy and power model in [2] for the on-chip interconnects and assume an activity factor of 5% for TDP (given that the network operates at 2 GHz and memory at 20 MHz). The benchmarks show an order of magnitude lower utilization of network. ADC/DAC energy and power are scaled for 5-bit and 2-bit precision [27]. While the state-of-the art ReRAM device supports 4 to 6 resistance levels [6], strong non-uniform analog resistance due to process variation makes it challenging to program ReRAM for analog convolution, resulting in convolution errors [12]. We conservatively limit the number of cell levels to two and use multiple cells in a row to represent one data.

Performance Model For determining the IMP performance, we develop a cycle accurate simulator which uses an integrated network simulator [22]. Note ReRAM array executes instructions in order, instruction latency is deterministic, network communication is rare, and compiler schedules instruction statically after accounting for network delay. Thus estimated performance for IMP is highly accurate.

7 Results

7.1 Configurations Studied

In this section we evaluate the proposed In-Memory Processor (**IMP**), and compare it to state-of-art **CPU** and **GPU** baselines. We use an Intel Xeon E5-2697 v3 multi-socket server as CPU baseline and Nvidia Titan XP as the GPU baseline. The IMP configuration (shown in Table 4) evaluated has 4,096 tiles and 64 128×128 ReRAM arrays in each tile.

Parameter	**CPU** (2-sockets)	**GPU** (1-card)	**IMP**
SIMD slots	448	3840	2097152
Frequency	3.6 GHz	1.58 GHz	20 MHz
Area	912.24 mm^2	471 mm^2	494 mm^2
TDP	290 W	250 W	416 W
Memory	7MB L2; 70MB L3 64GB DRAM	3MB L2 12GB DRAM	1GB RRAM

Table 5. Comparison of CPU, GPU, and IMP Parameters

Table 5 compares important system parameters of the three configurations analyzed. IMP has significantly higher degree of parallelism. IMP enjoys 546× (4681×) more SIMD slots than GPU (CPU). The massive parallelism comes at lower frequency, IMP is 80× (180×) slower than GPU (CPU) in terms of clock cycle period. IMP is approximately area neutral compared to GPU, and about 2× lower area than the 2-socket CPU system. The TDP of IMP is significantly higher, however we will show that IMP has lower average power consumption and energy consumption (Section 7.3).

7.2 Operation Study

Figure 7 presents the operation throughput of CPU, GPU, and IMP, measured by profiling microbenchmarks of add, multiply, divide, sqrt and exponential operations. We compile the microbenchmarks with -O3 option and parallelize it using OpenMP for the CPU. We find IMP achieves orders of magnitude improvement over the conventional architectures. The reason is two fold: massive parallelism and reduction in data movement. The proposed architecture IMP has 546× (4681×) more SIMD slots compared to GPU (CPU) as shown in Table 5. Although IMP has lower frequency, it more than compensates this disadvantage by avoiding data movement. CPU and GPU have to pay a significant penalty for reading the data out of off-chip memory and passing it along the on-chip memory hierarchy to compute units.

IMP speedup is especially higher for the simple operations. The largest operation throughput is achieved by addition (2,460× over CPU and 374× over GPU), which has smallest latency in IMP. On the other hand, division and transcendental functions take many cycles to produce the results. For example, it takes 62 cycles for division and 115 cycles for exponential, while addition takes only 3 cycles. Therefore, the throughput gain becomes smaller for complex operations. While CPU and IMP per-operation throughput reduces for higher latency operations, GPU throughput increases. This is because the GPU performance is bounded by the memory access time, and unary operators (exponential and square root) have less amount of data transfer from the GPU memory.

Figure 8 and 9 show the operation latency of addition and multiplication for different input size. We compare the execution time of single-threaded CPU, multi-threaded CPU

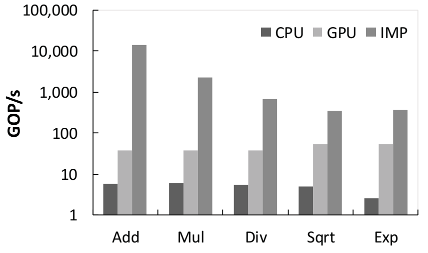

Figure 7. Operation throughput (log scale).

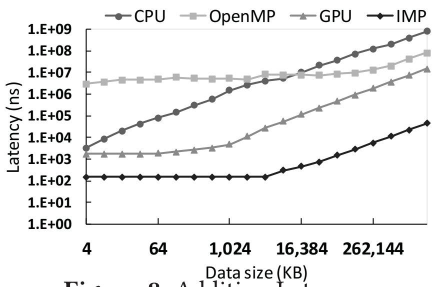

Figure 8. Addition Latency.

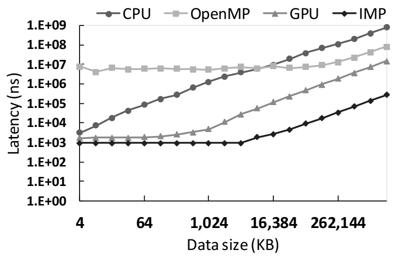

Figure 9. Multiplication Latency.

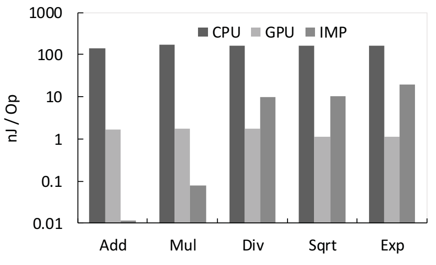

Figure 10. Operation energy.

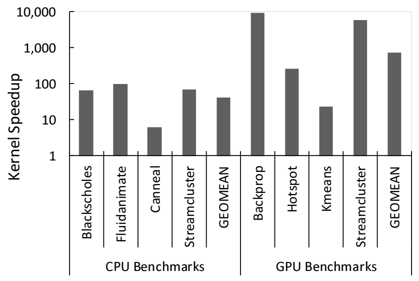

Figure 11. Kernel speedup.

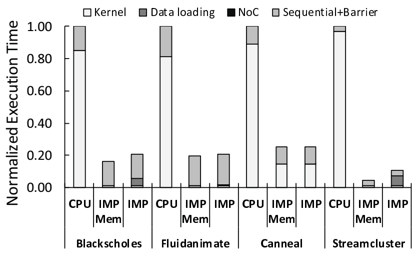

Figure 12. CPU Application performance.

(OpenMP), and GPU. IMP offers the highest operations performance among the three architectures, even for the smallest input size (4KB).

Figure 10 shows the energy consumption for each operation. Because of the high operation latency and the large energy consumption of ADC, we observe higher energy consumption for the complex operations relative to GPU. Ultimately, the instruction mix of the application will determine the energy efficiency of the IMP architecture.

7.3 Application Study

In this section we study the application performance. First, we analyze kernel performance shown in Figure 11. For CPU benchmarks, the figure shows performance for hot kernels in PARSEC benchmarks. We assume that non-kernel code of PARSEC benchmarks are executed in the CPU. Note that this data transfer overhead is taken into account in the results of IMP. The GPU benchmarks from Rodinia are relatively small, hence we regard them as application kernels. We observe a 41× speedup for CPU benchmarks and 763× speedups for GPU benchmarks.

GPU benchmarks obtain higher performance improvement in IMP because of the opportunity to use dot product operations and higher data level parallelism. On the other hand, the speedup for kmeans is limited to 23×. kmeans deals with Euclidean distance calculation of 34 dimensional vectors, and this incurs many element-wise multiplications. Although kmeans shows significant DLP available in the distance calculations, we could not fully utilize the DLP of the application because of the capacity limitation of the IMP's SIMD slots. This series of multiplications of distance calculation increases its critical latency and limits the speedup. As suggested in the operation throughput evaluation on Figure 7, IMP achieves higher performance especially when the kernel has significant DLP and many simple operations. We observe in general mul, add, and movl instructions are most common, while movg, reduce_sum and lut are less frequent. For example, a blackscholes kernel has 14% add, 21% mul, and 58% local move instructions. The rest are mask and lut.

The performance results for the overall PARSEC application are presented in Figure 12. For this result, we assume two scenarios: (1) IMP (memory) assumes IMP is integrated into the memory hierarchy and the memory region for the kernel is allocated in IMP. (2) IMP (accelerator) is a configuration when IMP is used as an accelerator and requires data copy as GPUs do. While we believe IMP (memory) is the correct configuration, IMP (accelerator) is a near-term easier configuration which can be a first step towards integrating IMP in host servers.

On average, IMP (accelerator) yields a 5.55× speedup and IMP (memory) provides 7.54× for the Region of Interest (ROI). We observe that 41× kernel speedup does not translate to similar application speedup due to Amdahl's law. Figure 12 also shows the breakdown of the execution time, which is divided into kernel, data loading, communication on NoC, and the non-kernel part of the ROI. The non-kernel part is mainly composed of time for barrier and unparalleled parts of the program. It can be seen that 88% of execution time can be off-loaded to IMP. We also observe that large fraction of the execution time on ReRAM is used for data loading (4× of the kernel at maximum). Thus, as suggested

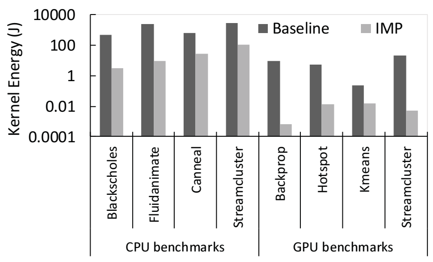

Figure 13. Energy consumption.

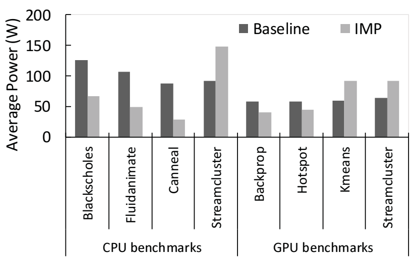

Figure 14. Average power.

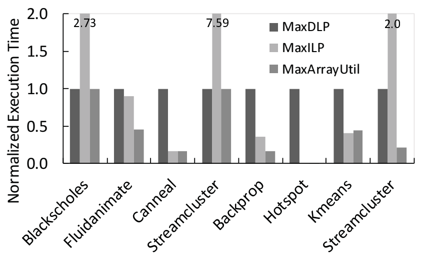

Figure 15. Compiler optimizations.

Config	Blackscholes	Fluidanimate	Canneal	Streamcluster	Backprop	Hotspot	Kmeans	Streamcluster
MaxDLP	665 / 1	1015 / 1	7220 / 1	2698 / 1	1028 / 1	1081893 / 1	3623 / 1	5386 / 1
MaxILP	377 / 5	437 / 9	1216 / 1212	159 / 129	184 / 32	3125 / 1024	134 / 38	287 / 257
MaxArrayUtil	665 / 1	437 / 4	1228 / 444	2698 / 1	171 / 27	1024 / 3125	1584 / 3	1169 / 6
Lifetime (years)	8.89	20.1	32.2	22.1	15.7	250	5.88	12.8

Table 6. (1) IB latency (cycles) and # of IBs for different optimization targets. (2) Lifetime

before, in-memory accelerator is better coupled with the existing memory hierarchy to avoid data loading overhead. We also find the NoC time is not the bottleneck, because of the efficient reduction scheme supported by the reduction tree network integrated in the NoC.

Figure 13 shows the total energy consumption of the entire application (thus includes both kernel and non-kernel energy for PARSEC). We find 7.5× and 440× energy efficiency for CPU benchmarks and GPU benchmarks, respectively. This energy reduction is partly due to energy efficiency of IMP for kernel's instruction mix and partly due to reduced execution time.

Figure 14 shows the average power consumption of the benchmarks. The TDP of IMP is high when compared to GPU and CPU (Table 5). ADCs are the largest contributer to peak power. The required resolution for ADCs is a function of maximum number of operands supported for *n*-ary instructions in our ISA. To contain the TDP, we limit the ADC resolution to 5-bits and thereby limiting the number of operands for *n*-ary instructions (`add`, `dot`). While this may affect the performance of a customized dot-product based machine learning accelerator significantly, it is not a serious limitation for general purpose computation. Although IMP's TDP is high due to the ADC power consumption, the average power consumption is dependent on the instruction's requirement for ADC resolution. For example, the ADCs consume less power for instructions with fewer operands. We find that the average resolution for ADC is 2.07 bit (maximum resolution is 5-bit). Overall, the average power consumption for IMP is estimated to be 70.1 W. The average power consumption measured for the benchmarks in the baseline is 81.3 W.

7.4 Effect of Compiler Optimizations

We introduce three optimization targets to the compiler and evaluate how each optimization affects the results. The first optimization target is **MaxDLP**, which creates one IB per module to maximize DLP. This policy is useful when the data size is larger than the SIMD slots IMP offers. However, the module does not have an opportunity to exploit ILP in the program. Also, IB expansion is not applied for this policy.

The second optimization target is **MaxILP**, which fully utilizes the ILP and lets IB expansion expand all multi-dimensional data in the module. This will create largest number of IBs per module and shortest execution time for single module. However, because of the sequential part of the IB, array utilization becomes lower. This policy can increase the overall execution time when the kernel is invoked multiple times due to insufficient SIMD slots in IMP.

The third optimization target, **MaxArrayUtil**, maximizes the array utilization considering the number of SIMD slots needed by input data. For example, if the incoming data consumes 30% of the total SIMD slots in IMP, each module can use 3 IBs to fully utilize all the arrays while avoiding multiple kernel invocations. The compiler optimizes under the constraint of maximum IBs available per module

Table 6 shows the maximum IB latency and the number of IBs per module. Figure 15 presents the execution time of different optimization policies normalized to MaxDLP (baseline). MaxArrayUtil represents the best possible performance provided by the compiler optimizations under resource constraints imposed by IMP. Overall it provides an average speedup of 2.3×.

Two other optimizations not captured by above graph are node merging and pipelining. On average, the module latency is reduced by 13.8% with node merging and 20.8% with pipelining.

7.5 Memory Lifetime

We evaluate the memory lifetime by calculating the write intensity of the benchmarks (last row in Table 6). Based on the assumption in [26], we consider the ReRAM cells to wear out beyond 10^{11} writes. The compiler balances the writes to the arrays by assigning and using ReRAM rows in a round-robin manner. Assuming the arrays are continuously used for kernel computation (but not while the host is processing), the median of expected lifetime is *17.9 years*.

8 Related Work

To the best of our knowledge this is the first work that demonstrates the feasibility of general purpose computing in ReRAM based memory. Below we discuss some of the closely related works.

ReRAM Computing Since ReRAMs have been introduced in [42], several works have leveraged its dot-product computation functionality for neuromorphic computing [25, 34]. Recently, ISAAC [39] and PRIME [13] use ReRAMs to accelerate several Convolutional Neural Networks (CNNs). ISAAC proposes a full-fledged CNN accelerator with carefully designed pipelining and precision handling scheme. PRIME studies a morphable ReRAM based main memory architecture for CNN acceleration. PipeLayer [41] further supports training and testing of CNN by introducing efficient pipelining scheme. Aside from CNN acceleration, ReRAM arrays have been used for accelerating Boltzmann machine [9] and perception network [45]. While it has been shown analog computation in ReRAM can substantially accelerate the machine learning workloads, none have targeted general purpose computing exploiting the analog computation functionality of ReRAM. Another interesting work, Pinatubo [28], has modified peripheral sense-amplifier circuitry to accomplish logical operations like AND and OR. While this approach appears promising to build complex arithmetic operations, doing arithmetic on multi-bit ReRAM cells using bitwise operations comes with several challenges. Orthogonal to this work, we extend the set of supported operations at low cost.

ReRAM has also been explored to implement logic using Majority-Inverter Graph (MIG) logic [7, 32, 40]. In this approach each ReRAM bit-cell acts as a majority gate. Since resistive bit-cell is acting as a logic gate, it cannot store data during computation. Let us refer to this approach as *ReRAM bit-cell as logic*. A critical difference between this approach and ours is that we leverage *in-situ* operations where operations are performed in memory over the bit-lines without reading data out. The ReRAM bit-cell as logic approach is a flavor of *near-memory* computing technique where input data is read out of memory and fed to another memory location which acts as a logic unit, thus requiring data-movement.

Furthermore, operations using majority gates can be extremely slow, requiring huge number of memory accesses to implement even simple functions. For example, a multiply is implemented using ≈56000 majority-gate operations (majority-gate operation requires one memory cycle) and 419 ReRAM cells [40]. Our approach implements a multiply in 18 memory cycles without requiring any additional ReRAM cells. While we demonstrated that IMP architecture/programming framework can work with large real-world general purpose data-parallel applications, ReRAM as logic approach [7] has been demonstrated for only sequential micro-kernels (e.g. hamming, sqrt, square etc) with no comparison to CPU or GPU systems.

Near-Memory Computing Past processing-in-memory (PIM) solutions move compute *near* the memory [4, 5, 10, 17, 18, 24, 31, 33, 35, 38, 46, 47]. The proposed architecture leverages an emerging style of in-memory computing referred to as bit-line computing [3]. Since, bit-line computing re-purposes memory structures to perform computation in-situ, it is intrinsically more efficient than near-memory computing which augments logic near memory. More importantly, it unlocks massive parallelism at near-zero silicon cost.

Recent works have leveraged bit-line computing in SRAM [3, 21, 23] and DRAM [37, 38]. These works have demonstrated only a handful of compute operations (bit-wise logical, match and copy) making them limited in applicability for general purpose computing. Furthermore this work is the first to develop a programming framework and compiler for in-memory bit-line computing. Our software stack can be utilized for leveraging bit-line computing in other memory technologies.

9 Conclusion

This paper proposed novel general-purpose ReRAM-based In-Memory Processor architecture (IMP), and its programming framework. IMP substantially improves the performance and energy efficiency for general-purpose data parallel programs. IMP implements simple but powerful ISA that can leverage the underlying computational efficiency. We propose the programming model and the compilation framework, in which users use TensorFlow to develop a program and maximize the parallelism using the compiler's toolchain. Our experimental results show IMP can achieve 7.5× over PARSEC CPU benchmarks and 763× speedup over Rodinia GPU benchmarks.

Acknowledgments

We thank members of M-Bits research group and the anonymous reviewers for their feedback. This work was supported in part by the NSF under the CAREER-1652294 award and the XPS-1628991 award, and by C-FAR, one of the six SRC STAR-net centers sponsored by MARCO and DARPA.

References

[1] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Man, Rajat Monga, Sherry Moore, Derek Murray, Jon Shlens, Benoit Steiner, Ilya Sutskever, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Oriol Vinyals, Pete Warden, Martin

Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. http://download.tensorflow.org/paper/whitepaper2015.pdf

[2] Nilmini Abeyratne, Reetuparna Das, Qingkun Li, Korey Sewell, Bharan Giridhar, Ronald G Dreslinski, David Blaauw, and Trevor Mudge. 2013. Scaling towards kilo-core processors with asymmetric high-radix topologies. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 496–507.

[3] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute Caches. In *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*. IEEE.

[4] 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. https://doi.org/10.1145/2749469.2750386

[5] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[6] Fabien Alibart, Ligang Gao, Brian D Hoskins, and Dmitri B Strukov. 2012. High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm. *Nanotechnology* 23, 7 (2012), 075201.

[7] Debjyoti Bhattacharjee, Rajeswari Devadoss, and Anupam Chattopadhyay. 2017. ReVAMP : ReRAM based VLIW Architecture for in-Memory comPuting. *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017* (3 2017), 782–787. https://doi.org/10.23919/DATE.2017.7927095

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* January (2008), 72–81. https://doi.org/10.1145/1454115.1454128

[9] M N Bojnordi and E Ipek. 2016. Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning. *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016), 1–13. https://doi.org/10.1109/HPCA.2016.7446049

[10] Jay B. Brockman, Shyamkumar Thoziyoor, Shannon K. Kuntz, and Peter M. Kogge. 2004. A Low Cost, Multithreaded Processing-in-memory System. In *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture (WMPI '04)*. ACM, New York, NY, USA, 16–22. https://doi.org/10.1145/1054943.1054946

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 44–54.

[12] Pai-Yu Chen, Deepak Kadetotad, Zihan Xu, Abinash Mohanty, Binbin Lin, Jieping Ye, Sarma Vrudhula, Jae-sun Seo, Yu Cao, and Shimeng Yu. 2015. Technology-design co-optimization of resistive cross-point array for accelerating learning algorithms on chip. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. IEEE, 854–859.

[13] Ping Chi, Shuangchen Li, and Cong Xu. 2016. PRIME : A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory. In *IEEE International Symposium on Computer Architecture*. IEEE, 27–39. https://doi.org/10.1109/ISCA.2016.13

[14] Marius Cornea, John Harrison, Cristina Iordache, Bob Norin, and Shane Story. 2000. Divide, Square Root, and Remainder Algorithms for the IA-64 Architecture. *Open Source for Numerics, Intel Corporation* (2000).

[15] John R. Ellis. 1986. *Bulldog: A Compiler for VLSI Architectures*. MIT Press, Cambridge, MA, USA.

[16] Milos Ercegovac, Jean-Michel Muller, and Arnaud Tisserand. [n. d.]. Simple Seed Architectures for Reciprocal and Square Root Reciprocal. ([n. d.]). http://arith.cs.ucla.edu/publications/Recip-Asil05.pdf

[17] A. Farmahini-Farahani, Jung Ho Ahn, K. Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*.

[18] Basilio B. Fraguela, Jose Renau, Paul Feautrier, David Padua, and Josep Torrellas. 2003. Programming the FlexRAM Parallel Intelligent Memory System. *SIGPLAN Not.* 38, 10 (June 2003), 49–60. https://doi.org/10.1145/966049.781505

[19] John Harrison, Ted Kubaska, Shane Story, et al. 1999. The computation of transcendental functions on the IA-64 architecture. In *Intel Technology Journal*. Citeseer.

[20] Miao Hu, R. Stanley Williams, John Paul Strachan, Zhiyong Li, Emmanuelle M. Grafals, Noraica Davila, Catherine Graves, Sity Lam, Ning Ge, and Jianhua Joshua Yang. 2016. Dot-product engine for neuromorphic computing. In *Proceedings of the 53rd Annual Design Automation Conference on - DAC '16*. ACM Press, New York, New York, USA, 1–6. https://doi.org/10.1145/2897937.2898010

[21] Supreet Jeloka, Naveen Akesh, Dennis Sylvester, and David Blaauw. 2015. A Configurable TCAM / BCAM / SRAM using 28nm push-rule 6T bit cell *(IEEE Symposium on VLSI Circuits)*.

[22] Nan Jiang, James Balfour, Daniel U Becker, Brian Towles, William J Dally, George Michelogiannakis, and John Kim. 2013. A detailed and flexible cycle-accurate network-on-chip simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 86–96.

[23] M. Kang, E. P. Kim, M. s. Keel, and N. R. Shanbhag. 2015. Energy-efficient and high throughput sparse distributed memory architecture. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*.

[24] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In *Proceedings of ISCA*, Vol. 43.

[25] Kuk-Hwan Kim, Siddharth Gaba, Dana Wheeler, Jose M Cruz-Albrecht, Tahir Hussain, Narayan Srinivasa, and Wei Lu. 2011. A Functional Hybrid Memristor Crossbar-Array/CMOS System for Data Storage and Neuromorphic Applications. *Nano Letters* 12, 1 (2011), 389–395. https://doi.org/10.1021/nl203687n

[26] J. B. Kotra, M. Arjomand, D. Guttman, M. T. Kandemir, and C. R. Das. 2016. Re-NUCA: A Practical NUCA Architecture for ReRAM Based Last-Level Caches. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 576–585. https://doi.org/10.1109/IPDPS.2016.79

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

[28] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 1–6.

[29] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. 1993. The multiflow trace scheduling compiler. *The Journal of Supercomputing* 7, 1 (01 May 1993), 51–142. https://doi.org/10.1007/BF01205182

[30] Martha Mercaldi, Steven Swanson, Andrew Petersen, Andrew Putnam, Andrew Schwerin, Mark Oskin, and Susan J Eggers. 2006. Instruction scheduling for a tiled dataflow architecture. In *ACM SIGOPS Operating*

Systems Review, Vol. 40. ACM, 141–150.

[31] Mark Oskin, Frederic T Chong, Timothy Sherwood, Mark Oskin, Frederic T Chong, and Timothy Sherwood. 1998. Active Pages: A Computation Model for Intelligent Memory. *ACM SIGARCH Computer Architecture News* 26, 3 (1998), 192–203. https://doi.org/10.1145/279358.279387

[32] P. E. Gaillardon, L. Amaru, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli. 2016. The Programmable Logic-in-Memory (PLiM) computer. *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2016), 427–432.

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

[34] Mirko Prezioso, Farnood Merrikh-Bayat, BD Hoskins, GC Adam, Konstantin K Likharev, and Dmitri B Strukov. 2015. Training and operation of an integrated neuromorphic network based on metal-oxide memristors. *Nature* 521, 7550 (2015), 61–64.

[35] S.H. Pugsley, J. Jestes, Huihui Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and Feifei Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*.

[36] Jury Sandrini, Marios Barlas, Maxime Thammasack, Tugba Demirci, Michele De Marchi, Davide Sacchetto, Pierre-Emmanuel Gaillardon, Giovanni De Micheli, and Yusuf Leblebici. 2016. Co-Design of ReRAM Passive Crossbar Arrays Integrated in 180 nm CMOS Technology. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 6, 3 (9 2016), 339–351. https://doi.org/10.1109/JETCAS.2016.2547746

[37] V. Seshadri, K. Hsieh, A. Boroum, Donghyuk Lee, M.A. Kozuch, O. Mutlu, P.B. Gibbons, and T.C. Mowry. 2015. Fast Bulk Bitwise AND and OR in DRAM. *Computer Architecture Letters* (2015).

[38] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. [n. d.]. RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*.

[39] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, and Rajeev Balasubramonian. 2016. ISAAC : A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (6 2016), 14–26. https://doi.org/10.1109/ISCA.2016.12

[40] Mathias Soeken, Saeideh Shirinzadeh, Luca Gaetano, AmarÃž Rolf, and Giovanni De Micheli. 2016. An MIG-based Compiler for Programmable Logic-in-Memory Architectures. *Proceedings of the 2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)* 1 (2016). https://doi.org/10.1145/2897937.2897985

[41] L. Song, X. Qian, H. Li, and Y. Chen. 2017. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 541–552. https://doi.org/10.1109/HPCA.2017.55

[42] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. 2008. The missing memristor found. *Nature* 453, 7191 (2008), 80.

[43] Pascal O Vontobel, Warren Robinett, Philip J Kuekes, Duncan R Stewart, Joseph Straznicky, and R Stanley Williams. 2009. Writing to and reading from a nano-scale crossbar memory based on memristors. *Nanotechnology* 20, 42 (2009), 425204.

[44] Z. Wei, Y. Kanzawa, K. Arita, Y. Katoh, K. Kawai, S. Muraoka, S. Mitani, S. Fujii, K. Katayama, M. Iijima, T. Mikawa, T. Ninomiya, R. Miyanaga, Y. Kawashima, K. Tsuji, A. Himeno, T. Okada, R. Azuma, K. Shimakawa, H. Sugaya, T. Takagi, R. Yasuhara, K. Horiba, H. Kumigashira, and M. Oshima. 2008. Highly reliable TaOx ReRAM and direct evidence of redox reaction mechanism. In *2008 IEEE International Electron Devices Meeting*. IEEE, 1–4. https://doi.org/10.1109/IEDM.2008.4796676

[45] Chris Yakopcic and Tarek M Taha. 2013. Energy efficient perceptron pattern recognition using segmented memristor crossbar arrays. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*. IEEE, 1–8.

[46] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*.

[47] Qiuling Zhu, B. Akin, H.E. Sumbul, F. Sadi, J.C. Hoe, L. Pileggi, and F. Franchetti. 2013. A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing. In *3D Systems Integration Conference (3DIC), 2013 IEEE International*.