In-Memory Data Parallel Processor

Daichi Fujiki Scott Mahlke Reetuparna Das
University of Michigan
{dfujiki, mahlke, reetudas}@umich.edu

1 Overview

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 [1, 2]. Despite 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 be configured for diverse applications. We combat this problem by proposing a programmable in-memory processor architecture and programming framework.

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, so as to expose the above benefits of ReRAM computing to general purpose data parallel programs.

We design a compute stack for ReRAM-based in-memory computing by addressing problems in each layer of the stack. We design a processor architecture that re-purposes resistive memory to support data-parallel in-memory computation. We extend the ReRAM memory array to support in-situ operations beyond the dot product and design a simple ISA with basic compute capability. We develop a compiler that transforms *Data Flow Graphs* (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/con-straints of ReRAM-based computation. Although our 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.

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

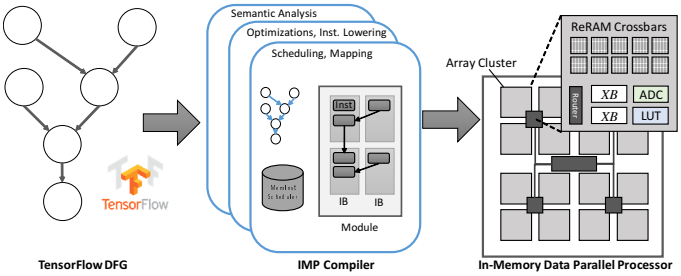

Figure 1: Architecture Overview.

2 In-Memory Data Parallel Processor

2.1 Processor Architecture / ISA

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. 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 proposed architecture extends the ReRAM array to support in-situ operations beyond dot product. In addition to the computation primitives supported by previous works of ReRAM based accelerators (i.e. addition and dot-product), we demonstrate ReRAM arrays can implement subtraction by current draining, and element-wise multiplication by introducing column Digital-Analog-Converters.

The Instruction Set Architecture (ISA) for in-memory computation 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 of the architecture. Compared to a standard SIMD ISA, In-memory ISA only supports four compute instructions (i.e. add, sub, dot, and mul) and 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. Control flow is facilitated via condition computation and selective moves.

2.2 Programming Model / Compiler

We choose Google's TensorFlow as the programming front-end for proposed in-memory processor. TensorFlow is a pop-

ular programming model for machine learning. We observe that TensorFlow's programming semantics is a perfect marriage of data-flow and vector-processing (or SIMD) that can be applied to more general applications. By using TensorFlow, programmers write the kernels which will be offloaded to the memory.

TensorFlow expresses the kernel as a *Data Flow Graph* (DFG). Data-flow explicitly exposes the Instruction Level Parallelism (ILP) in the program. TensorFlow's DFG 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. TensorFlow exposes Data Level Parallelism (DLP) explicitly as vector processing. Since there is no subscript notation in TensorFlow, 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). Thus, our compiler can infer the memory access pattern easily and enjoy more optimization opportunities.

We develop a *TensorFlow compiler* that generates binary code for our in-memory data-parallel processor. Overall compilation flow is shown in Figure 2. The TensorFlow 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 compiler generates a module by unrolling a single dimension of multi-dimensional input vectors. 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.

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 in their native ISA 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 discussed 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 ISA.

Our compiler explores several interesting optimizations. For example, node merging exposes multi-operand compute primitives of ReRAM by merging several DFG nodes and reduces intermediate writes. IB expansion decomposes multi-dimensional vectors operands and expands them into several instruction blocks (IBs) with lower-dimension vectors to further exploit ILP and DLP. The compiler then performs VLIW style scheduling which maximizes ILP within a module while minimizing communication between arrays.

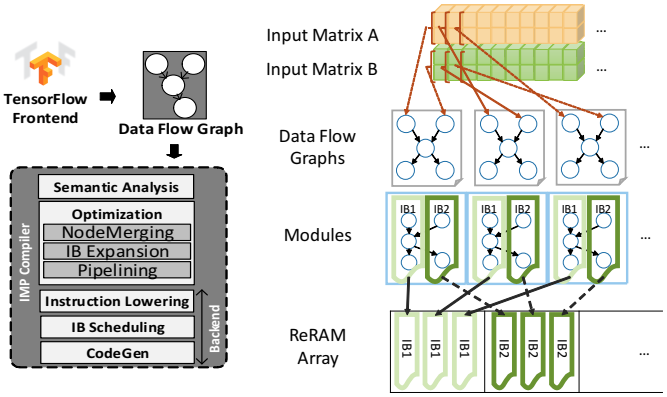

Figure 2: Compiler Flow (Left). Execution Model (Right).

2.3 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.

Each module is composed of one or more Instruction Blocks (IBs). 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, as a phase of its optimization step.

3 Results

We identified kernels within a variety of parallel application from PARSEC multi-threaded CPU benchmarks and Rodinia GPU benchmarks, and rewrote them in TensorFlow. Our experimental results show that the proposed architecture provides a speedup of 763× and energy gains of 230× over a server class GPU (Titan-Xp) for the Rodinia benchmarks. For CPU PARSEC benchmarks, we are able to offload hot kernels and execute 87% of the program directly in-memory, leading to an overall speed-up of 7.5×. The proposed architecture operates with a thermal design power (TDP) of 415 W, and reduces the average power by 1.26×. Assuming the arrays are continuously used for kernel computation (but not while the host is processing), we estimated the expected lifetime of ReRAM to be *17.9 years*.

References

[1] Chi et al. 2016. PRIME : A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory. *ISCA* (2016).

[2] Shafiee et al. 2016. ISAAC : A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. *ISCA* (2016).