Exploiting Sparsity, Compression and In-Memory Compute in Designing Data-Intensive Accelerators

by

Thomas Chih Chen

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering)
in the University of Michigan
2019

Doctoral Committee:

Associate Professor Zhengya Zhang, Chair
Assistant Professor Ronald G. Dreslinski
Professor Michael P. Flynn
Professor Wei Lu

Thomas Chih Chen

tcchen@umich.edu

ORCID iD: 0000-0003-3168-9122

ACKNOWLEDGEMENTS

First, I sincerely thank my advisor Prof. Zhengya Zhang, for his guidance, help, and support over the years. His advice and encouragement have inspired me to overcome greater challenges and deeply influenced my research. Prof. Zhangs consideration and good insights have developed many students professionally, with broader views and maturity. All of us appreciate his dedication to the group. I would like to thank Prof. Wei Lu, Prof. Michael Flynn, Prof. Ronald Dreslinski for being part of my committee. They gave many constructive suggestions on my research projects and provided valuable advice for my dissertation. I would also like to thank Prof. Mark Kushner and Prof. Olshausen for sharing insightful ideas and extending my research toward a wider range of applications.

My research could not be completed without the help and support from the sponsors: DARPA (HR0011-17-2-0018), the SONIC Center, Intel, and NSF (GRFP DGE 1256260), and the University of Michigan. I would like to thank them for sponsoring my works.

I am thankful for the discussions and collaborative efforts with my lab mates Phil Knag, Jung Kuk Kim, Chia-Hsiang Chen, Yun Sung Park, Shuanghong Sun, Yaoyu Tao, Ching-En Lee, Jacob Botimer, Shiming Song, Chester Liu, Sung-Gun Cho, Teyuh Chou, and Reid Pinkham. Working with them has been an invaluable part of my years at the University of Michigan. Their insights and support have helped me grow as a researcher. I appreciate those whom I know in SI program, MICL, EECS, and other engineering departments for their support. I value the interactions and friendships I have made here.

Finally, I would like to thank my parents for their continuous support. I would also like to thank my siblings and relatives who have been around and cheering me up over the years. Again, I appreciate the great individuals at the University of Michigan, sponsors, friends, colleagues, and family who have influenced me to complete this dissertation.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Enabled by technology scaling, processing parallelism has been continuously increased to meet the demand of large-scale and data-intensive computations. However, the effort to increase processing parallelism is largely hindered by the von Neumann bottleneck. To achieve a higher performance, domain-specific computing has become the most promising direction. Domain-specific computing employs highly optimized datapaths, simplified control and efficient dataflow to enable the dense integration of processing elements with optimized memory access. Many domain-specific designs have demonstrated significantly better figure of merit than a general-purpose CPU or GPU, but the von Neumann bottleneck still limits the maximum achievable performance.

To reduce the data transfer cost, a closer integration between memory and computation is needed, which ultimately leads to the so-called in-memory computing approach. In-memory computing re-purposes memory cell array for multiply-accumulate operations and apply both bit-line and word-line parallelism to realize large matrix computation before memory readout, eliminating the von Neumann bottleneck entirely. However, in-memory computing is inherently analog compute, where limited precision and high sensitivity to noise pose major challenges.

This thesis work presents two approaches to address the von Neumann bottleneck: 1) reducing the amount of data that needs to be moved by sparsity and data compression; and 2) robust multi-bit in-memory compute design to extend the applicability of in-memory compute to a wider range of applications.

With video input and 3D features, a video processor requires many times larger memory size and computation than a 2D image processor. In this work, I chose a video sequence inference processor to demonstrate sparsity-oriented optimizations using a quantized all-spiking network, where the sparsity can reach a high 90% level. By kernel compression and activation compression, memory size can be reduced further by 43% and 64%, respectively. High data sparsity and memory compression lead to two orders of magnitude of improvement in performance and energy. The design was demonstrated in a 2.53mm^2 40nm CMOS chip for video sequence inference that achieved 1.70TOPS with a power dissipation of 135mW at 0.9V and 250MHz. The results show the effective use of sparsity and data compression to loosen the von Neumann bottleneck.

It is well known that in-memory computing is limited in operand and output precision, which restricts its applications to binary or low-precision applications. Through an algorithmic transformation using a residual approach, I demonstrate that it is possible to map a high-precision partial differential equation (PDE) solver to a low-precision 5-bit in-memory computing. To support multi-bit computation, I adopt both width and level modulation of word-line pulses. To reduce the cost and improve the speed of analog-to-digital conversion, I employ a compact array of dual-ramp single-slope (DRSS) ADCs for bit-line readout. These ideas were demonstrated in a 1.87mm^2 180nm test chip made of four 320×64 multiply-accumulate (MAC) SRAMs, each supporting 128× parallel 5b×5b MACs with 32 5-bit output ADCs and consuming 16.6mW at 200MHz. The prototype was able to solve a 127×127 PDE grid at 56.9 GOPS. This SRAM based in-memory compute provides over 40× compute density than an equivalent ASIC, demonstrating that the von Neumann bottleneck can be removed for applications that require higher precisions.

This work shows the importance of algorithm-architecture-circuit co-design for uncovering opportunities to mitigate and remove the von Neumann bottleneck. The design techniques and approaches can be applicable to a wide array of applications for improving performance and efficiency.

CHAPTER I

Introduction

The vast majority of modern computer processors are designed based on the von Neumann architecture [7] as shown in Fig. 1.1 that roughly consists of four parts: a processing unit for arithmetic and logic operations, a control unit that steps through instructions, a memory that stores data and instructions, and input and output devices. The architecture has served us well in the era when semiconductor devices are relatively large, expensive and slow, and the datasets are small.

With the rapid scaling down of CMOS devices following Moore's law [8], CMOS devices have become exponentially smaller, cheaper, and faster. Processors can afford to have multiple or many processing cores [9]. Graphics processing units (GPU) and application-specific integrated circuits (ASIC) can provide hundreds or thousands of processing units on chip to boost performance [10, 11]. Enabled by the rapid growth of processing capabilities, new classes of applications, such as computer vision [12], natural language processing [13], autonomous navigation [14], virtual reality [15] and augmented reality [16], crypto currency [17] and blockchain [18], have emerged. A commonality among all these emerging applications is that the datasets can be massive and the compute models are large and complex.

An example is shown in Fig. 1.2 for the state-of-the-art deep neural network (DNN) models [2] compiled by Canziani, et al. A state-of-the-art DNN model contains over

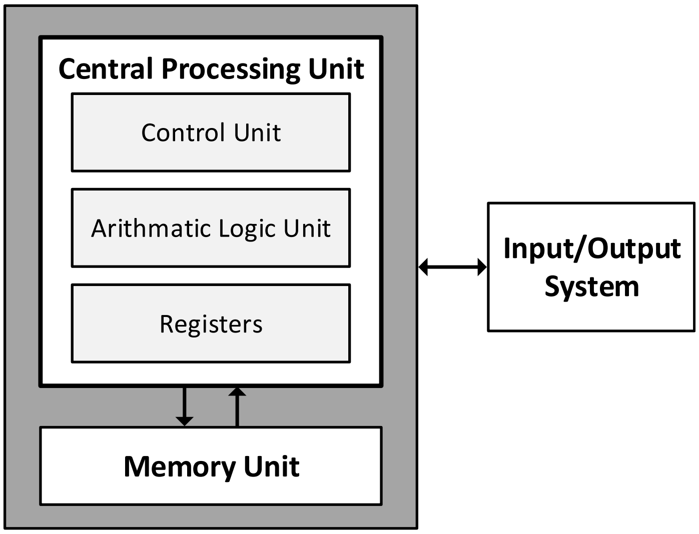


Figure 1.1: Von Neumann architecture for computer processors. [1]

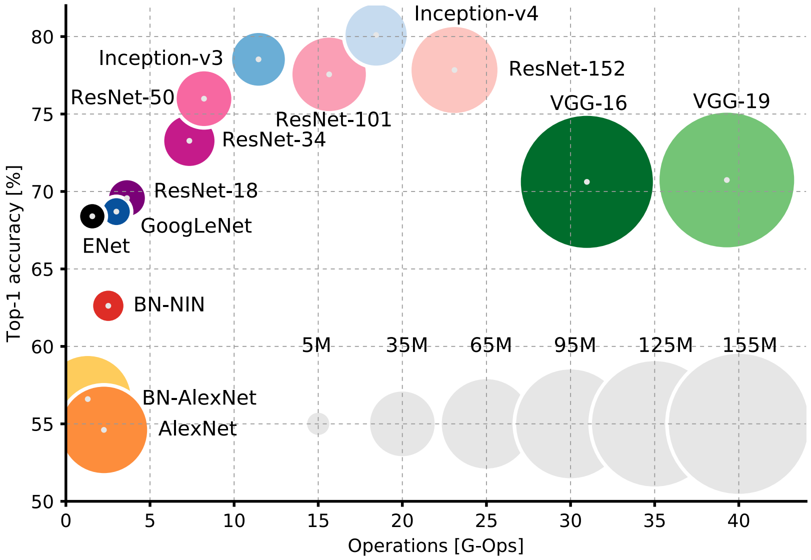


Figure 1.2: Top-1 accuracy vs operations, parameters for state-of-the-art DNN architectures. The size of each blob is proportional to number of parameters, as indicated in the legend at bottom right corner. [2]

Table 1.1: Estimated energy cost for data access and floating-point computation on 45nm technology. [6]

FP Add	
16 bit	0.4 pJ
32 bit	0.9 pJ
FP Mult	
16 bit	1 pJ
32 bit	4 pJ

SRAM Cache (64 bit)	
8 KB	10 pJ
32 KB	20 pJ
1 MB	100 pJ
DRAM	1.3-2.6 nJ

100 million parameters, and requires 40 GOPs to process a relatively low-resolution image of 224×224×3 (3 channels).

One can argue that the current CMOS device technology can comfortably provide the level of parallelism needed for the growing list of emerging applications. For example, Nvidia's V100 GPU accelerator already boasts a 15 TFLOPS performance [19]. However, as Horowitz famously pointed out, the compute problem we are facing is no longer how parallel we can make the processing units, but how much it costs to supply the input data and take away the output data from the processing units [6].

Indeed, as Table 1.1 shows, in a 45nm CMOS technology, moving data from a relatively small-sized 8KB SRAM cache memory costs 10× more in energy than a 16-bit floating-point multiplication. The energy cost escalates by another 10× when moving data from a larger 1MB SRAM. If data have be moved from off-chip DRAM, we can expect to pay yet another 10× energy cost. The lopsided energy breakdown shows that data movement, rather than parallel processing, is the determining factor in high-performance and efficient processor designs. This is known as the von Neumann bottleneck [20]. It is not surprising that the vast majority of the Nvidia V100 GPU's 300 W thermal design power is attributed to the memory interface [19].

1.1 Related Work and Challenges

Architecture specialization for a domain of applications has been named as one promising way forward [21]. Domain-specific architectures are designed for special classes of applications. Compared to general-purpose processors that incur a high control overhead, a less efficient processing pipeline, and frequent load and store accesses, domain-specific architectures can be made much more efficient by simplifying the control, optimizing the data flow, and maximizing local data reuse.

Google's tensor processing unit (TPU) [3] is a prime example of a domain-specific processor that is tailored to DNN workloads. The core of the TPU is a 256×256 8-bit vector-matrix product engine. The 64k multiply-accumulate (MAC) units are wired up in a systolic array. The partial results are passed between one MAC unit to its nearest neighbor, allowing the final results to be accumulated along the compute path and avoiding expensive SRAM access. The massive parallelism, together with an optimized data flow, contribute to the TPU's record performance and efficiency: using only half the silicon area of the Intel Haswell CPU or Nvidia Kepler K80 GPU, and half the power, TPU provides 25× more MAC operations [22] than both.

Domain specialization allows the TPU to score a much better figure of merit than a general-purpose CPU and GPU. However, TPU is still hurt by the von Neumann bottleneck as evidenced in the roofline plot shown in Fig. 1.3. A roofline curve shows the performance of a processor as the data reuse is increased. A roofline curve has two distinct parts, a slanted part and a flat part. The flat part defines the maximum achievable performance of the processor (the performance roof); and in the slanted part, the performance of the processor is capped by memory bandwidth. As data reuse is increased, the same memory bandwidth can support a higher performance. Fig. 1.3 clearly shows that the TPU raised the performance roof compared to a CPU and a GPU, but it also features a long slanted region. Many workloads, such as long short-term memory (LSTM) and multilayer perceptron (MLP), are still limited by

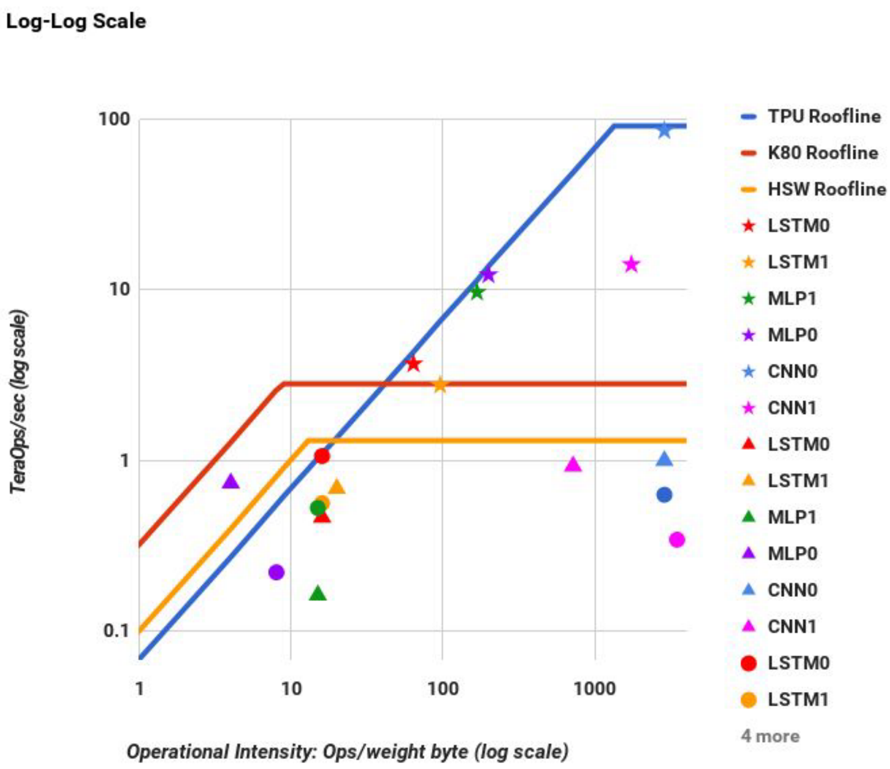


Figure 1.3: Roofline performance model for CPU, GPU, and TPU. [3]

memory bandwidth.

To alleviate or resolve the von Neumann bottleneck, near-memory compute and in-memory compute have been proposed. In near-memory compute, processing is moved very close to memory to reduce the data movement cost [23]. 3D integration technology is a key enabler of near-memory compute. High bandwidth memory (HBM) [24] and hybrid memory cube (HMC) [25] are schemes that stack layers of DRAM on top of a logic layer. The layers are interconnected by a large number of through-silicon vias (TSVs). Making use of the third dimension increases the memory capacity, and TSVs provide short-distance and dense connectivity to memory.

Based on HMC, two near-memory designs, Neurocube [4], shown in Fig. 1.4, and TETRIS [26], have been demonstrated. Both designs place processing units on the bottom logic layer, which access layers of DRAMs directly on top. Benefiting from 3D integration and the substantially reduced wiring distance and increased bandwidth,

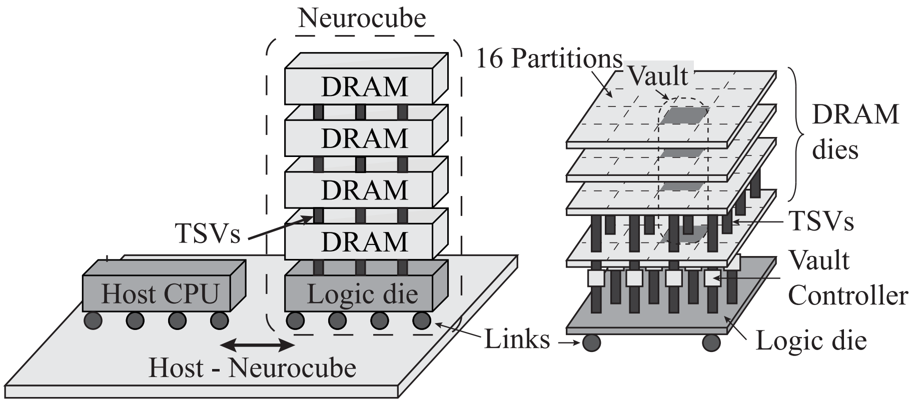


Figure 1.4: Neurocube architecture. [4]

Neurocube demonstrated 4× improvement in power efficiency compared to a GPU [4].

Note that near-memory compute still follows the von Neumann architecture with a much relaxed memory access bottleneck. In comparison, in-memory compute, illustrated in Fig. 1.5, relies on memory for both storage and compute. Compute is performed inside memory cells, without moving data out of memory. As a result, in-memory compute can offer a higher efficiency than near-memory compute. In-memory compute also activates multiple or all the word lines of a memory array in parallel to unleash the array's intrinsic high parallelism.

In-memory compute has been demonstrated in a number of silicon prototypes. Early work used SRAM for in-memory compute [27]. Lately, prototypes have been designed based on resistive RAM (RRAM) [28]. Comparing RRAM to SRAM, RRAM offers denser and nonvolatile storage, and it is the more attractive option for in-memory compute. However, SRAM is made of CMOS devices, which are more mature and can be readily demonstrated. SRAM-based in-memory compute accelerators, such as Conv-RAM [29], demonstrated more than 16× improvement in energy efficiency compared to a conventional digital accelerator.

In-memory compute is no longer limited by the von Neumann bottleneck. However, in-memory compute is fundamentally a form of analog compute that relies on

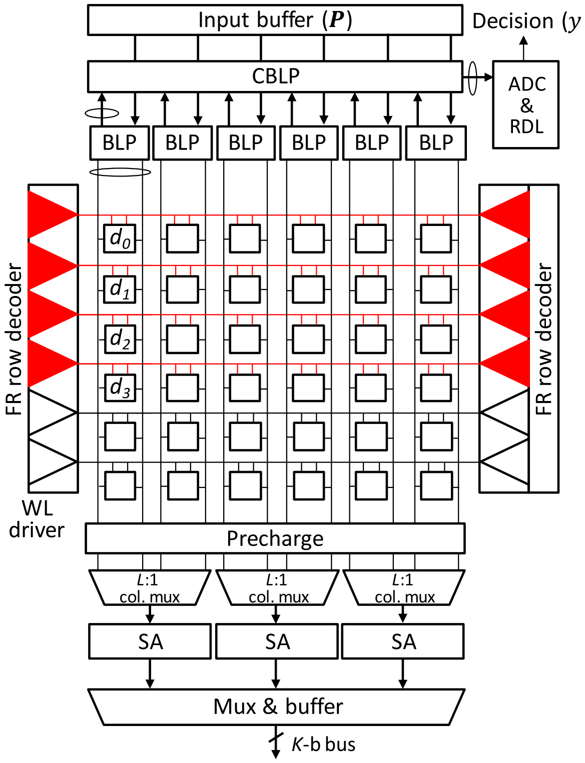


Figure 1.5: Deep in-memory computing architecture. [5]

modulating analog voltages and summing analog currents. As such, in-memory compute is less reliable than conventional digital compute, and it can be easily affected by noise, variations and offsets. Although in-memory compute can be made highly parallel, it is difficult to obtain high-precision results. Early efforts have relied on converting the results to 1 bit [30]. With proper circuit techniques, up to 7 bits can be obtained [29], but it requires costly analog-to-digital conversion. To sum up, limited precision and high sensitivity to noise and variation are the key challenges of in-memory compute.

1.2 Thesis Contribution

This thesis work provides new solutions targeting high-performance and energy-efficient accelerator design for data-intensive applications. As the datasets are large, the von Neumann bottleneck presents a major challenge. The proposed solutions are based on two primary approaches: 1) reducing the amount of data that need to be moved by exploiting sparsity and data compression; and 2) robust multi-bit in-memory compute design to extend the applicability of in-memory compute to a wider range of applications.

1.2.1 Data Sparsity

Sparsity is often inherent in large datasets. For example, sensory data can be highly sparse in the sense that the amount of meaningful information is low relative to the large number of raw data points. Even if the raw data appear dense, through a signal processing technique called compressed sensing [31], the raw data can be projected to a new space to make the data appear sparse. Popular compression techniques make use of this principle. For example, audio and image compression often employs discrete cosine transform (DCT) that expresses a finite sequence of data points in terms of a sum of cosine functions at different frequencies [32]. After applying DCT, audio and image data become sparse and can be efficiently compressed.

The brain does an amazing job in compressing input sensory data. The sensory inputs, e.g., images, videos, audio, speech, are coded in highly sparse neuron spikes for cognitive processing [33]. It is hypothesized that the brain employs an efficient coding scheme that maximizes both coding accuracy and sparsity [34], akin to compressed sensing. The high sparsity could be a key factor behind the ultra-high efficiency of the brain.

The data sparsity inspires the design of better accelerators. Sparsity implies that most of the data are zeros, leading to reduced workload and higher performance. More importantly, sparsity results in a lower memory traffic, which alleviates the von Neumann bottleneck.

In this work, a video sequence inference processor design is chosen to demonstrate data-sparsity-optimized design techniques. A video sequence inference processor takes

2D + time video inputs and extracts spatio-temporal features [35]. The extracted spatio-temporal features are used to infer the activities, or actions that are present in the input video. The video sequence inference task is highly challenging because the data size is significantly larger than 2D image processing that is commonly used in benchmarking machine learning hardware. However, video inputs are highly sparse, possibly more sparse than 2D images. By applying a neuro-inspired compressed sensing algorithm, the video inputs are efficiently coded in the feature domain. The data sparsity can reach a high 90% level, offering an opportunity to obtain both high performance and high efficiency.

Chapter 2 will discuss the design of a sparse, all-spiking accelerator for the implementation of a video sequence inference accelerator. This work was a joint effort with my group member Ching-En Lee. Sparsity is inherently high in videos; and sparsity can be further increased using a residual approach and a rectification technique. A residual approach operates on the incremental changes in each iteration, rather than the raw data directly. It is reasonable to expect that the residuals are sparser than the raw data. Rectification is an approximate compute technique that quantizes intermediate data to a few finite levels, allowing the intermediate data to be sparsified.

Backed by the high data sparsity and simple data compression techniques, the performance and energy of the accelerator can be improved by nearly two orders of magnitude. The substantial improvements are attributed to the effective use of data sparsity that loosens the von Neumann bottleneck.

1.2.2 Multi-Bit In-Memory Compute Accelerator

A key limitation of in-memory compute is the limited precision. Many prior designs resorted to storing 1-bit operands in memory or converting outputs to 1 bit, so as to match a digital memory with binary sense amplifiers and avoid costly ADCs.

The low precision of in-memory compute limits its applicability to simple or toy examples, e.g., binary neural networks [36], true or false classifications [37].

A partial differential equation (PDE) solver is chosen to demonstrate the multi-bit in-memory compute technique. A PDE solver is widely used in scientific applications. A PDE solver is commonly implemented using Jacobi method [38]. The equation and the solution space are first discretized to a grid, and the solutions are found through iterations of matrix-vector products.

PDE solver is a big-data workload: discretization results in a large number of data points. If an accurate solution is required, which is often the case, a fine grid of fine step size and a floating-point quantization are used, producing even more data. It can take tens of thousands of iterations to converge, requiring a large amount of data to be passed back and forth between memory and processing units. The von Neumann bottleneck becomes the limiter. Although in-memory compute can cut the data movement cost, it appears to be a mismatch for a PDE solver application due to the PDE's high precision requirement.

Chapter 3 will discuss how a residual approach is applied to reduce the precision requirement of a PDE solver, allowing a high-precision PDE solver can be mapped to a low-precision in-memory compute. The key is a residual technique that operates on the incremental differences of data between iterations rather than the raw data. Although the raw data can be in high precision, the incremental differences of the data between iterations are much smaller in magnitude and can be represented using a low precision. A drawback of the approach is that the convergence speed worsens with reduced precision. However, there exists an optimal point in the trade-off between]hlprecision and convergence speed. The optimal point for the PDE solver is near 4 to 5 bits, where the increase in convergence latency is still manageable.

Even after the PDE solver computation is quantized to 5 bits, it is still not feasible to map the computation to an existing in-memory compute accelerator. Some in-memory compute accelerators support only 1-bit multiplicands [27, 29, 30], 1-bit multipliers [30, 39], or 1-bit outputs [27, 30, 39], which are not sufficient for a PDE solver. None of the existing in-memory compute accelerators provides enough number of multi-bit ADCs to support the bandwidth needed for iterative solution updates in a PDE solver.

This work presents a number of new circuit approaches to enable a practical multi-bit in-memory compute. First, both width and level modulation are used in word line modulation to allow the multiplication of multi-bit operands. A delayed-locked loop (DLL) is used to generate well-control word line pulses, while offering tolerance against process voltage temperature (PVT) variations. Second, a dual-ramp single-slope (DRSS) ADC is used to perform the analog-to-digital conversion of bit line outputs. A DRSS ADC shortens the conversion time and it can be constructed using a shared centralized reference generation and compact comparator circuitry per column. The circuit techniques are demonstrated in an SRAM-based prototype. Over 40× compute density can be achieved compared to a conventional digital ASIC design.

A part of this work is contributed by my group member, Jacob Botimer. He has contributed to the design of the MAC SRAM module, and the implementation of the PDE solver chip.

This thesis work demonstrates that by exploiting sparsity, compression and multi-bit in-memory computing, it is possible to drastically reduce and eliminate data movement and overcome the von Neumann bottleneck. This work also points out the need to perform extensive analyses of the algorithms and models to uncover algorithm-architecture-circuits co-design opportunities. The results of this work have been demonstrated through silicon prototypes and verified by experimental measurements. The approaches and techniques can be applicable to a wide array of high-performance and efficient designs of future data-intensive accelerators.

CHAPTER II

Design of Video Sequence Inference Processor

2.1 Introduction

Object detection in videos is employed in a wide range of applications from smart user interface to surveillance and autonomous navigation. Due to the demanding resolution and frame rate of videos, real-time object detection has been a challenge. Designing real-time object detection on embedded platforms is especially difficult due to the limited energy source available on embedded platforms.

State-of-the-art object detection accelerators [40–44] have been designed based on SIFT [45], SURF [46] and DPM [47] algorithms. These popular algorithms extract 2D features from images, and compare them with features stored in a database [45,46] or perform classifications [47] on the features to recognize objects. The accelerators target real-time videos, but the base operations are on 2D images.

Video sequence classification, or action classification, operates on sequences of video frames to extract activity or action information from videos. Video sequence classification relies on extracting spatio-temporal features and performing classification on the spatio-temporal features, thus it is expected to demand more computation than the 2D processing of videos.

Classic video sequence classification relies on engineered features, such as cuboid [48], space-time Harris [49], and Hessian [50]. Each feature selection is tailored to a specific task, but may not deliver the best performance for every task. It is desirable to use automatically learned features that are most suitable for the data. An auto-encoder is one such approach that automatically learns sparse, shift-invariant spatio-temporal features [51]. Sparse coding is a similar approach that adapts an overcomplete dictionary of space-time functions (features) to represent time-varying natural images with high sparsity [35]. The space-time features resemble the motion-selective receptive fields (RF) of simple cells in the mammalian visual cortex, suggesting that the approach may be at work in the visual cortex [35].

In this work, we adopt a sparse coding approach called locally competitive algorithm (LCA) [52]. LCA is formulated as a compressed sensing method. When applied to videos, LCA learns the spatio-temporal RFs (STRFs) and encodes inputs using a sparse set of STRFs. As such, LCA is highly effective in reducing the input size, allowing the most salient STRFs to be extracted for classification.

LCA can be mapped to a spiking recurrent neural network (RNN) [53, 54]. Implemented using iterative forward projection and backward reconstruction, a video sequence inference processor based on spiking RNN can extract spatio-temporal RFs (STRFs), i.e., spatio-temporal features, from videos. The extracted STRFs can in turn enable action classification [55] and motion tracking [56] tasks.

Due to the large video data size, spatio-temporal, and iterative processing, the computational requirement of the video sequence inference RNN is high. Even for a relatively small-scale processing of a 6×6×64 video slice using 192 STRFs costs 200M multiply-accumulates (MACs). To enable a practical implementation, we adopt a residual formulation of the RNN [57] and apply an algorithm transformation by rectifying the residuals after each inference iteration to ternary spikes without costing classification accuracy. After the transformation, the intermediate data through the compute stages, i.e., activations and residuals, become spikes with a sparsity level well above 90%. The transformed algorithm leads to a sparse, all-spiking video inference processor design that reduces the computational complexity from 200M MACs to 4M select-accumulates per iteration, making it possible to support video processing in real-time at a reasonable power consumption. To reduce the large on-chip storage, we apply non-uniform delta encoding on the highly redundant STRFs and compressed column storage (CCS) on the highly sparse activations to reduce their memory size by 43% and 64%, respectively.

The design is demonstrated in a 2.53mm^2 40nm inference SoC that integrates a video sequence inference core and an OpenRISC core. The chip is measured to achieve 1.70TOPS at 0.9V and 250MHz, dissipating 135mW at room temperature. With the video sequence inference core extracting the activation response of STRFs, a soft-max classifier programmed on the OpenRISC core achieves a 76.7% classification accuracy on the 6-class KTH Human Action Dataset [58].

This work was a joint effort with my group member Ching-En Lee. My key contributions to this work were the algorithm analysis, architectural sparsity evaluation and the back-end layout and integration of the SoC.

The rest of the paper is organized as follows. Section II provides an overview of the baseline inference algorithm, and Section III shows how the algorithm is transformed to a sparse, all-spiking formulation to reduce its implementation cost. Section IV presents the design details of each compute layer and memory, and quantifies the performance and energy gain. Section V shows the chip implementation and measured results, and Section VI concludes the work.

2.2 Video Inference Algorithm Formulation

In this work, we adopt the LCA algorithm [52] to perform compressed sensing of videos. LCA can be mapped to a recurrent network of spiking leaky integrate-and-fire neurons, where a neuron's potential increases due to input excitation, and decreases due to inhibition by neighboring neurons. The LCA algorithm is described

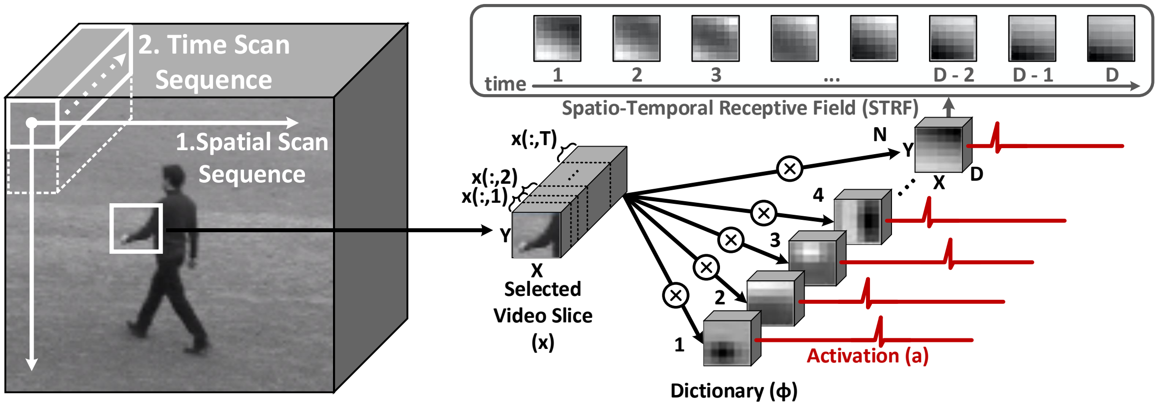


Figure 2.1: Illustration of video inference processing.

by equation (2.1).

$$
\begin{aligned}
\Delta \mathbf{u} &= \eta \Big[\mathbf{\Phi}^T \mathbf{x} - \big(\mathbf{\Phi}^T \mathbf{\Phi} - \mathbf{I} \big) \mathbf{a} - \mathbf{u} \Big] \qquad (2.1) \\
\mathbf{a} &= T_\lambda(\mathbf{u}),
\end{aligned}
$$

where $\mathbf{u}$ is the neuron potential, $\Delta\mathbf{u}$ is the potential update; η is the update step size; $\mathbf{\Phi}$ is the receptive fields (RF) of neurons, also known as the dictionary; $\mathbf{x}$ is the input; $\mathbf{a}$ is the neuron activation; and $\mathbf{I}$ is the identity matrix. $T_\lambda()$ is a binary threshold function and it outputs 1 if its input exceeds λ, or 0 otherwise. Dictionary $\mathbf{\Phi}$ and threshold λ are trained by stochastic gradient descent, which aims to maximize encoding accuracy and the sparsity of neuron activations.

In performing inference on video inputs, an input is divided to 3D segments for processing. In (2.1), $\mathbf{x}$ is a time series of T number of $X \times Y \times D$ consecutive and overlapping video segments, as shown in Fig. 2.1. The dictionary $\mathbf{\Phi}$ is a collection of N RFs, and each RF is a $X \times Y \times D$ spatio-temporal feature, known as STRF. $\mathbf{u}$, $\Delta\mathbf{u}$, and $\mathbf{a}$ are collections of N neurons' potentials, potential updates, and activations, respectively, over T time steps. Mathematically, $\mathbf{x}$ is a $V \times T$ matrix, where $V =$

XYD; $\mathbf{\Phi}$ is a $V \times N$ matrix; $\mathbf{u}$, $\Delta\mathbf{u}$ and $\mathbf{a}$ are $N \times T$ matrices.

The inference described by equation (2.1) consists of four functional steps:

1. *Charge*: Input $\mathbf{x}$ is projected to the feature space as described by $\mathbf{\Phi}^T\mathbf{x}$. The projection can be understood as encoding the input $\mathbf{x}$ in STRFs, i.e., extracting STRFs from the input. The projection increases, or charges, the neuron potential.

2. *Compete*: To maintain sparse activation, active neurons suppress other neurons. The inhibition weight between a pair of neurons is computed by correlating their STRFs, i.e., $\mathbf{\Phi}^T\mathbf{\Phi}$. Self-inhibition is removed by subtracting $\mathbf{I}$. The closer the two neurons' STRFs, the stronger the inhibition between the two neurons. Neuron activations trigger inhibitions as described by $\left(\mathbf{\Phi}^T\mathbf{\Phi} - \mathbf{I}\right)\mathbf{a}$.

3. *Leak*: Neuron potential decreases over time, and the leakage is proportional to the potential.

4. *Activate*: Neuron potential is thresholded to generate binary spikes.

The four steps above constitute one iteration of inference. Given an input $\mathbf{x}$, the inference is done by iterating the four steps until convergence. It is common to use a fixed number of iteration I. The baseline implementation is outlined in Fig. 2.2, where the leak step is omitted for simplicity.

The implementation complexity of one iteration of inference is analyzed and the results are listed in Table 2.1. The dictionary storage requires VN entries. The inhibitory weights are computed by $\mathbf{\Phi}^T\mathbf{\Phi} - \mathbf{I}$, requiring N^2V MACs. The N^2 inhibitory weights can be computed once and stored in memory.

In every iteration of inference, the charge step requires NVT MACs. Because the two inputs $\mathbf{\Phi}^T$ and $\mathbf{x}$ to the charge step do not change between iterations, the charge is computed only once per inference regardless of the number of iterations.

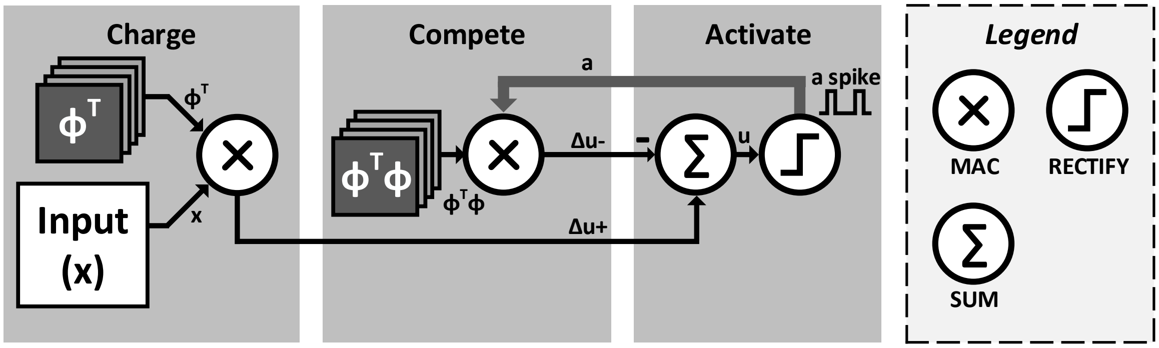


Figure 2.2: Baseline implementation of video inference.

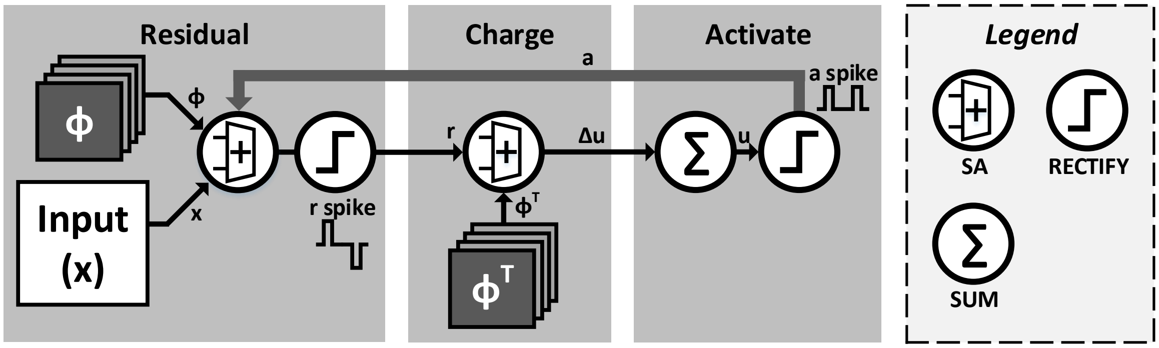
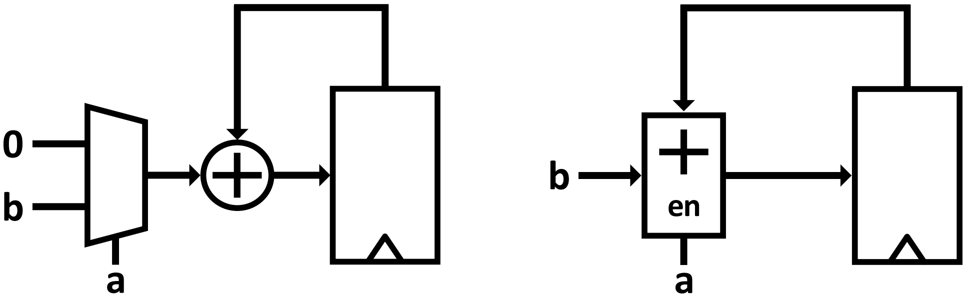


Figure 2.3: Sparse, all-spiking implementation of video inference.



(a)

(b)

Figure 2.4: SA implementations: (a) select-add, (b) skip-add.

Table 2.1: Baseline Implementation Complexity of One Iteration of Inference

Function	Storage (# weights)	Compute (# MACs)
Dictionary storage	VN	-
Inhibitory weight storage	N^2	-
Charge step	-	NVT
Compete step	-	N^2TI
Total	$VN + N^2$	$NVT + N^2TI$

Table 2.2: Implementation Complexity of One Iteration of Inference Using Residual Approach

Function	Storage (# weights)	Compute (# MACs)
Dictionary storage	VN	-
Residual step	-	$NVTI$
Charge step	-	$NVTI$
Total	VN	$2NVTI$

Table 2.3: Implementation Complexity of One Iteration of Inference Using Sparse and All-Spiking Approach

Function	Storage (# weights)	Compute (# SAs)
Dictionary storage	VN	-
Residual step	-	$NVTS_aI$
Charge step	-	$NVTS_rI$
Total	VN	$NVT(S_a + S_r)I$

The compete step is driven by neuron activations, requiring N^2T MACs per iteration for I iterations.

Typically the number of neurons (N) ranges from hundreds and more for practical applications, and video inference can be particularly challenging due to its large dimensionality and real-time processing requirement. A silicon implementation requires a large area and power.

2.3 Sparse and All-Spiking Inference Formulation

Video data is large, but it also contains high redundancy, especially from frame to frame. The redundancy offers opportunities for significant complexity reduction in storage and compute. The sparse coding algorithm also lends itself to an efficient implementation by exploiting its inherent sparsity.

We formulate the algorithm such that all steps operate on spiking inputs. As a result, expensive MACs are replaced by efficient select-accumulates (SAs); and operations are skipped if no spikes are present.

2.3.1 Rectification and Sparsification

The LCA equation can be reformulated by factoring the term $\mathbf{\Phi}^T$ in (2.1):

$$\begin{aligned} \Delta\mathbf{u} &= \eta\Big[\mathbf{\Phi}^T\left(\mathbf{x}-\mathbf{\Phi}\mathbf{a}\right)+\mathbf{a}-\mathbf{u}\Big] \\ \mathbf{a} &= T_\lambda(\mathbf{u}). \end{aligned} \tag{2.2}$$

The reformulated inference, first proposed by [57], can be interpreted as having four steps: residual, charge, leak and activate. The leak and activate steps are identical to the original formulation. The residual and charge steps are described below.

1. *Residual*: The input $\mathbf{x}$ is reconstructed, $\hat{\mathbf{x}} = \mathbf{\Phi}\mathbf{a}$. The reconstruction is subtracted from the input to obtain the residual $\mathbf{r} = \mathbf{x} - \hat{\mathbf{x}}$.

2. *Charge*: The residual is projected to the feature space, $\mathbf{c} = \mathbf{\Phi}^T\mathbf{r}$.

The residual form removes the storage of inhibitory weights and replaces it by computing the weights on the fly. As a result, the storage required is smaller, but the compute complexity poses a challenge, as shown in Table 2.2. To reduce the complexity, we propose to quantize the residuals. If the residuals can be quantized to binary spikes (1, -1), the computational complexity of the charge layer can be significantly simplified. However, as Fig. 2.5(a) shows, the binary quantization has a large impact on the classification accuracy when the activation density is low. With 0 being the binary threshold, small noise values near 0 are amplified, preventing convergence and degrading accuracy.

To fix this problem, we propose a min/max rectification to the residuals to quantize the residuals to ternary spikes. The residual rectification is done by applying thresholds of λ_r and $-\lambda_r$ to quantize the residuals to 1 (above λ_r), 0 (between -λ_r and λ_r), and -1 (below -λ_r). With appropriate threshold choices, the ternary quantization outperforms binary quantization by a large margin and can even match the unquantized accuracy, as shown in Fig. 2.5(a). The updated equation is given in (2.3), where T_{λ_r} is the min/max rectification function.

$$\begin{aligned} \Delta\mathbf{u} &= \eta\left[\mathbf{\Phi}^T T_{\lambda_r}\left(\mathbf{x} - \mathbf{\Phi}\mathbf{a}\right) + \mathbf{a} - \mathbf{u}\right] \\ \mathbf{a} &= T_{\lambda}(\mathbf{u}). \end{aligned} \tag{2.3}$$

A key advantage of quantizing the residuals to binary or ternary spikes is that the multiplication by these quantized values and accumulating the partial sums no longer requires a MAC. Instead, a simpler SA can be used. Suppose a is binary (0 or 1), multiplying a by b followed by accumulation can be done using an SA that is implemented as in Fig. 2.4(a), where a is used as the select input in the multiplexer to choose whether 0 (if a is 0) or b (if a is 1) is accumulated by the adder. The accumulated sum is saved in a register. Alternatively, SA can be implemented using a skip-add shown in Fig. 2.4(b), where a is used as the enable input to the adder to decide whether to accumulate b (if a is 1) or not (if a is 0). Although the example was shown for the binary spike case, the implementation can be easily modified to support ternary spikes.

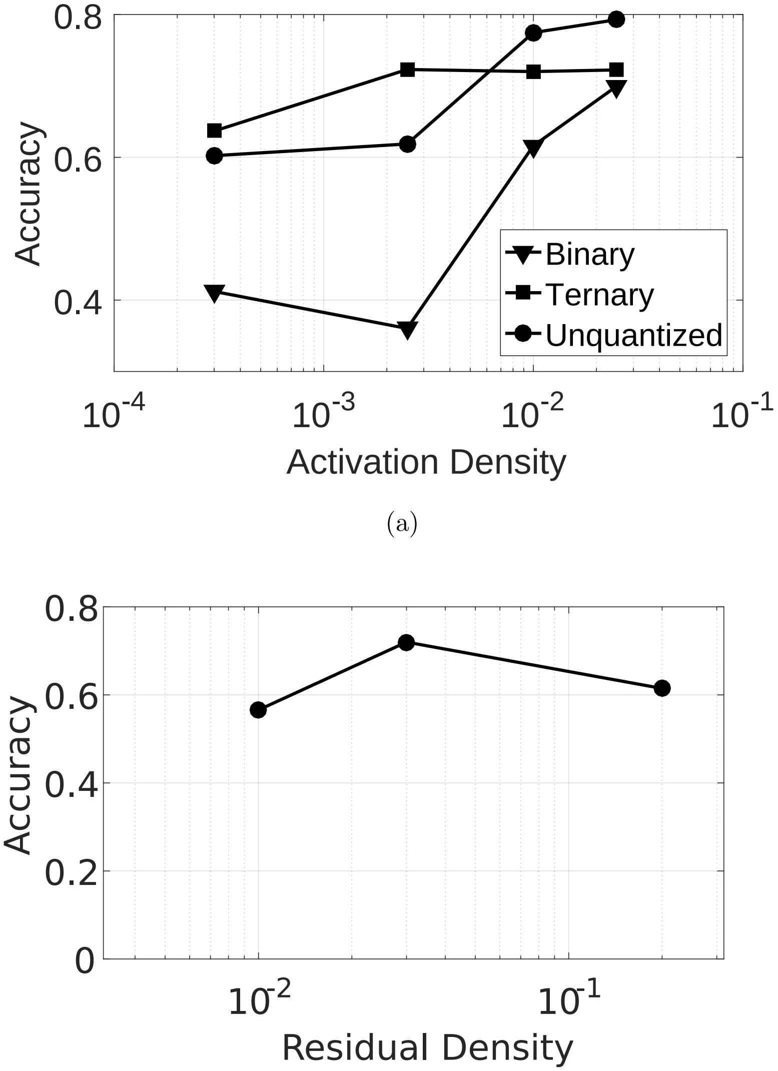
Similar to the residual rectification, neuron activation is obtained by rectifying neuron potentials to produce sparse, binary spikes. Binary spikes allow the reconstruction in the residual step to be implemented using SAs, presenting another opportunity for significant complexity and power reduction.

Taking advantage of both residual rectification and neuron activation, the sparse, all-spiking approach can be implemented as shown in Fig. 2.3. It features a lower complexity compared to the conventional residual approach as summarized in Table 2.3, where S_a and S_r refer to the density, or fraction of nonzero entries, in neuron activations and the residuals, respectively. The sparser the inputs (i.e., the lower density), the less the amount of effectual workload. However, sparsifying the inputs (activations or residuals) can degrade the classification accuracy. The effects are illustrated in Fig. 2.5. The activation density $S_a = 1\%$ and residual density $S_r = 3\%$ are nearly optimal for the KTH Human Action Dataset [58]. Below or above the optimal density results in under- or over-representation of the input, and degradation in classification accuracy.

2.3.2 Design Specification and Parameter Settings

We present a prototype video inference processor to demonstrate the sparse, all-spiking LCA approach. The prototype design, including the model and parameters, is based on the KTH dataset [58]. The inference processor takes video inputs in 6×6×64 slices, and divides into 57 6×6×8 ($T = 57$, $V = 6\times6\times8 = 288$) consecutive and overlapping segments for processing.

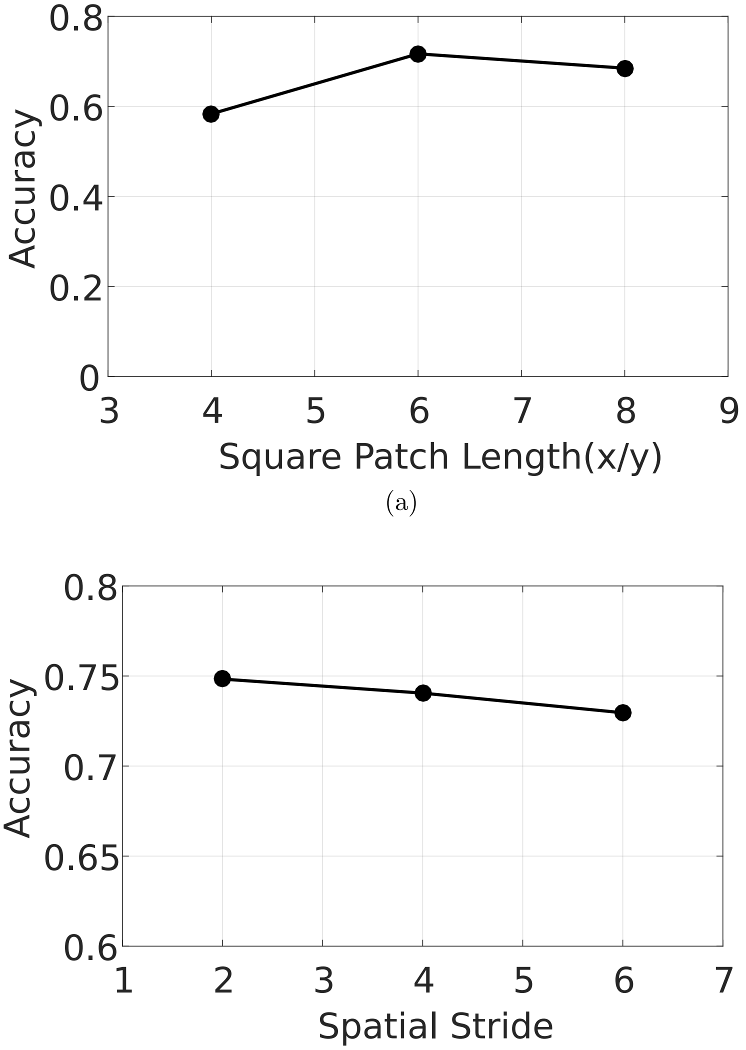
The optimal X-Y patch size is determined by the size of features for a dataset. For KTH dataset, 6×6 patch size provides the best accuracy as shown in Fig. 2.6(a). More spatial overlap (smaller spatial stride) produces better results as shown in Fig. 2.6(b).



(a)

(b)

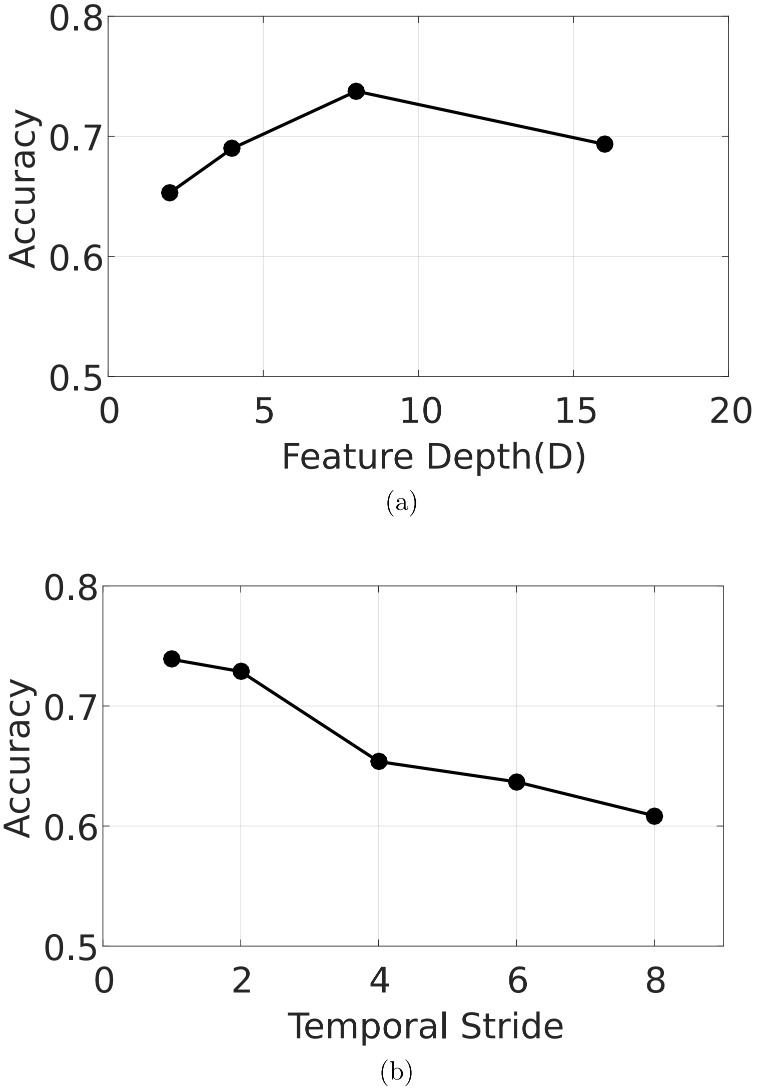
Figure 2.5: Effect of a) activation density and b) residual density on classification accuracy.



(a)

(b)

Figure 2.6: Effect of a) patch size and b) 6×6 patch spatial stride on classification accuracy.



(a)

(b)

Figure 2.7: Effect of a) feature depth and b) time overlap on classification accuracy.

However, in the prototype design, we chose no overlap to reduce the processing complexity. It degrades accuracy by only 2%.

The optimal STRF depth is determined by the action sequence duration for a dataset. For the KTH dataset, a larger depth yields better accuracy as shown in Fig. 2.7(a). We used a depth of 8, below which the accuracy drops by about 2% per depth reduction of 1. Temporal overlap (small temporal stride) is essential for guaranteeing a good accuracy, e.g., increasing the temporal stride from 1 to 4 reduces the accuracy by more than 8% as shown in Fig. 2.7(b). In the prototype design, we

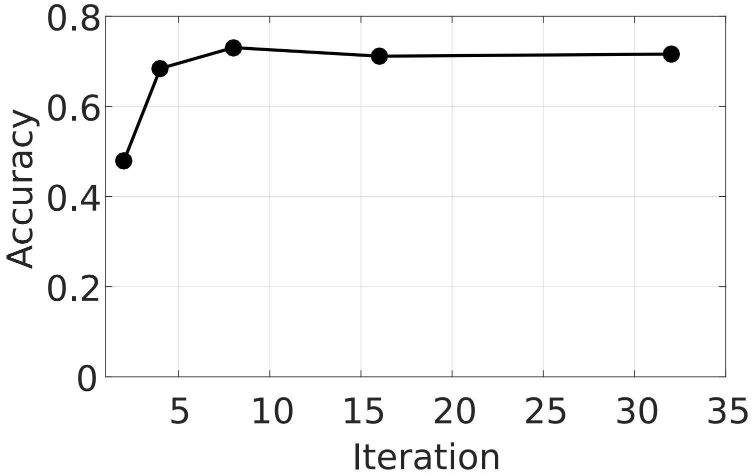


Figure 2.8: Effect of number of iterations on classification accuracy.

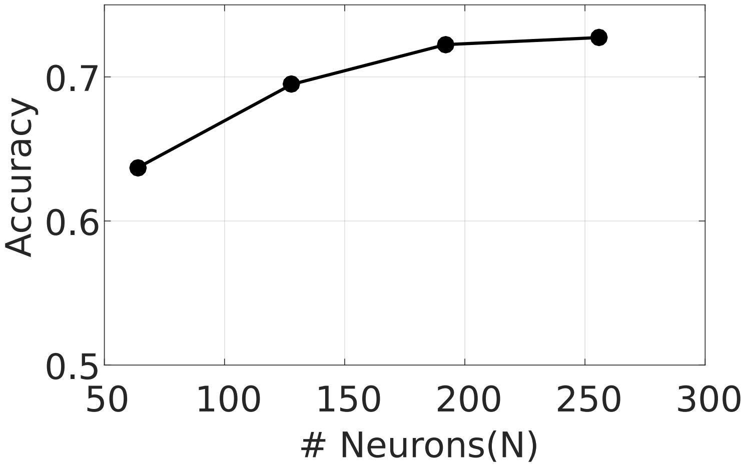


Figure 2.9: Effect of number of neurons on classification accuracy.

chose a stride of 1.

The number of neurons, i.e., the number of STRFs, is dependent on the input size and it affects the classification accuracy as shown in Fig. 2.9. In testing the prototype design, we employ 192 neurons ($N = 192$). Each neuron's STRF is sized 6×6×8. The STRF weights are quantized to 8 bits. Simulations show that 6 to 8 iterations are sufficient in Fig. 2.8, beyond which the accuracy saturates. We used 8 iterations (I = 8) for measurement in this work. Based on the STRFs extracted from the video, action classification can be performed.

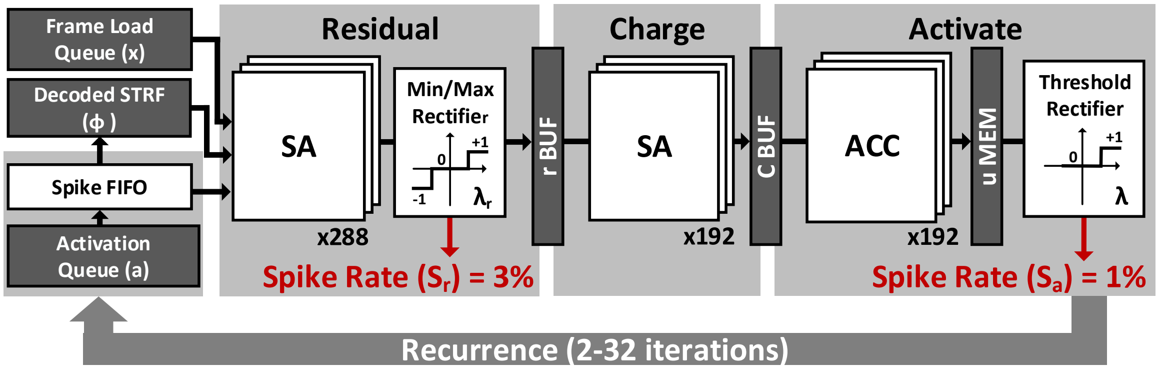


Figure 2.10: Architectural sketch of 3-layer implementation of video inference processor.

To realize this prototype chip, 54KB memory is needed to store the dictionary. The density of neuron activations and residuals are optimally set to $S_a = 1\%$ and $S_r = 3\%$, respectively. The sparse, all-spiking approach reduces the number of operations per inference from 200M MACs to 4M SAs, which translates to a significant reduction in complexity and power consumption.

2.4 Design of Video Inference Processor

The video inference processor is made of three compute layers: residual layer, charge layer, and activate layer as illustrated in Fig. 2.10. Each layer corresponds to one step outlined in the previous section (the leak step is absorbed as part of the charge layer). The residual and charge layers are the workhorse of the inference processor. The inputs to the residual layer are sparse binary neuron spikes. The inputs to the charge layer are sparse residuals in the form of ternary spikes. Inputs are streamed through the three layers and back to the residual layer for the next iteration.

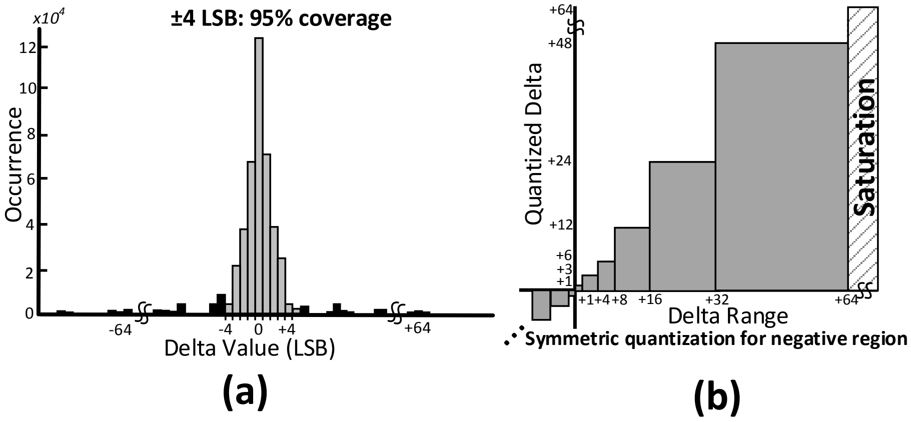


Figure 2.11: (a) Distribution of deltas between frames of STRFs; (b) non-uniform quantization of deltas.

2.4.1 Dictionary Compression and Non-Uniform Quantization

The dictionary $\mathbf{\Phi}$ and its transpose $\mathbf{\Phi}^T$ are accessed by the residual layer and the charge layer, respectively. Since the two layers operate concurrently in a streaming pipeline and the dictionary elements' access orders are different, both $\mathbf{\Phi}$ and $\mathbf{\Phi}^T$ are stored on chip, requiring 108KB of memory for the prototype design. Due to the high access bandwidth needed for highly parallel processing, the dictionary memory is divided into banks, sacrificing the storage efficiency. The dictionary memory alone is estimated to take 2mm^2 chip area in a 40nm CMOS technology.

In the prototype design, each dictionary element is a 6×6×8 8-bit STRF that is essentially a sequence of 8 6×6 frames. Redundancy exists between consecutive frames, making it possible to compress each STRF to save memory, chip size and power. In Fig. 2.11(a), we plot the distribution of the pixel-by-pixel differences between consecutive frames of STRFs that are learned by training on the KTH dataset. The results show that 95% of the pixel-by-pixel differences cover a narrow range of only 4 LSBs.

The similarity between consecutive frames motivates the delta encoding of STRFs by storing the first 6×6 8-bit frame as the anchor frame, and subsequent frames as

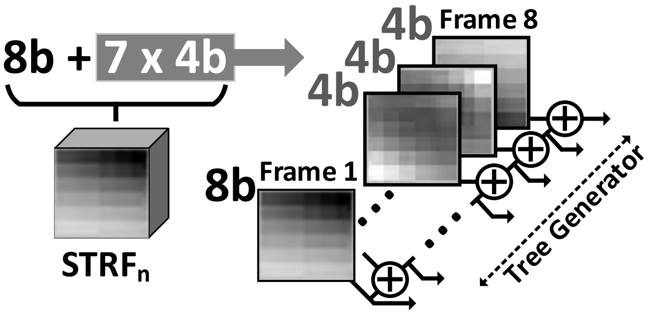


Figure 2.12: a tree generator for decompressing delta-encoded STRF.

4-bit deltas to the previous frame. The delta encoding reduces the dictionary storage by 43%.

Although 4 bits are sufficient to cover 95% of the deltas, a better result requires a larger coverage. To keep deltas to 4 bits while increasing the range of coverage, we propose the non-uniform quantization of deltas as shown in Fig. 2.11(b). The non-uniform quantization is specifically tailored to the delta distribution: smaller quantization step sizes are used at the lower end, and increasingly larger quantization step sizes are used towards the higher end to keep the number of quantization steps to 15.

The delta-encoded dictionary elements need to be decompressed before being used in compute. We employ a tree generator, shown in Fig. 2.12 to take the anchor frame as the base, and sequentially add the deltas to recover the remaining frames. With delta encoding and taking into account the overhead of tree generator, the dictionary memory storage in our prototype design, including compression and decompression, occupies 27% less area compared to the baseline.

2.4.2 Residual Layer

The residual layer computes the reconstruction $\hat{\mathbf{x}}$ $(V \times T)$ by multiplying $\mathbf{\Phi}$ $(V \times N)$ by $\mathbf{a}$ $(N \times T)$. Recall that since $\mathbf{a}$ consists of binary activations, the matrix multiplication is done by SAs. The input $\mathbf{a}$ is provided to the residual layer one

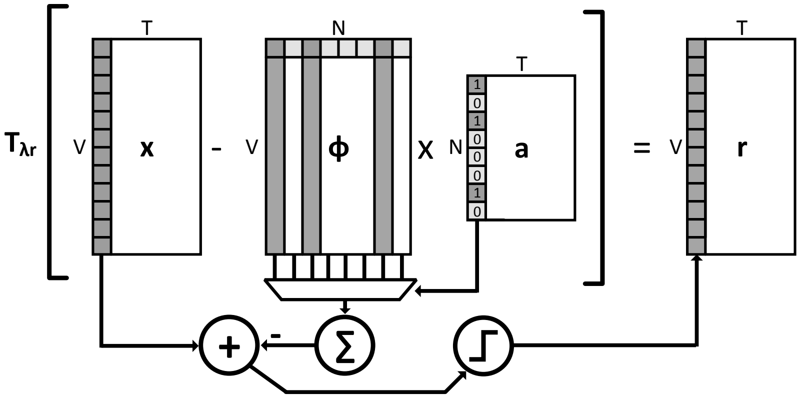


Figure 2.13: Visualization of residual compute.

column at a time, as the time-series output of N neurons from the activate layer. Since activations are sparse, we use a spike detector to skip 0 activations and provide the addresses of the activated neurons.

The residual layer computation is illustrated in Fig. 2.13. For each column of **a**, the spike detector looks at a block of entries at a time and finds the address of the first entry that is 1. Suppose in processing column i of **a**, the spike detector outputs j as the first entry in column i that is 1, then column j of $\mathbf{\Phi}$ is read from memory, decompressed by the tree generator, and accumulated by the SA array as the temporary output of column i of $\hat{\mathbf{x}}$. We employ an array of V SAs to compute one vector accumulation at a time. The process continues with the spike detector providing the next nonzero entry. Upon completion, the reconstruction is subtracted from the input **x**; and the results are rectified to obtain the residuals. Since the reconstruction is computed column by column, the residuals are obtained column by column and provided to the charge layer in this order.

An implementation of the residual layer is shown in Fig. 2.14. The number of actual accumulations done by the SA array is $NVTS_a$, with S_a being the density of 1's in **a**. Since V SAs operate in parallel, the residual layer takes on average NTS_a $= 192\times57\times1\% = 109$ cycles.

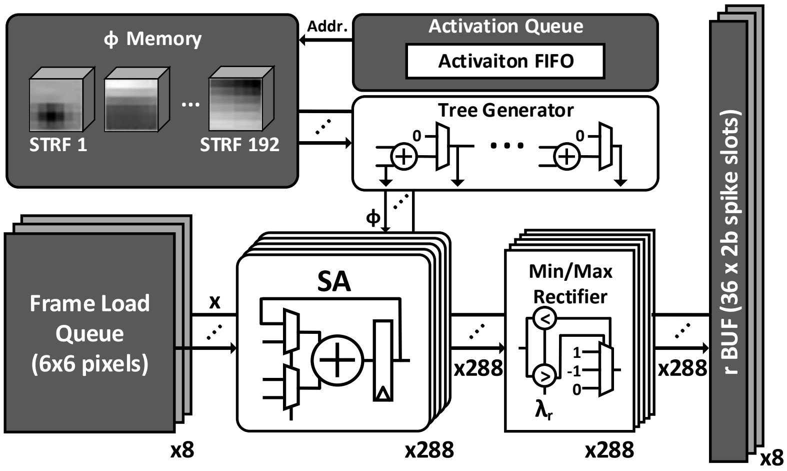


Figure 2.14: Residual layer design.

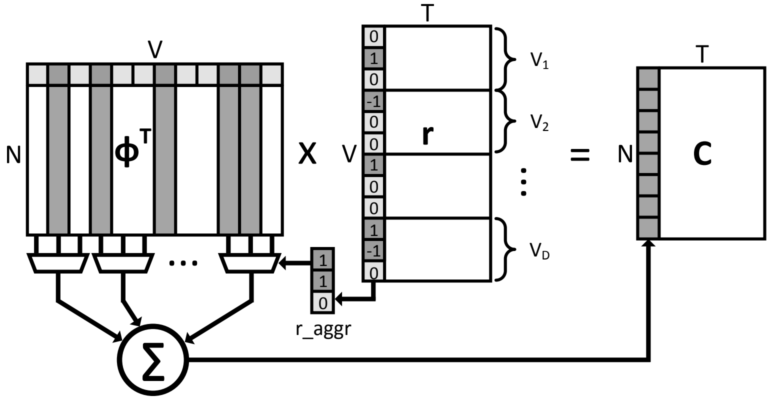


Figure 2.15: Visualization of charge compute.

2.4.3 Charge Layer

The charge layer computes the charge $\mathbf{c}$ by multiplying $\mathbf{\Phi}^T$ $(N \times V)$ by $\mathbf{r}$ $(V \times T)$. Since $\mathbf{r}$ is a collection of ternary spikes {0, -1, 1}, the matrix multiplication is also done by SAs.

A similar architecture as the residual layer can be designed to implement the charge layer. The input $\mathbf{r}$ is provided one column at a time, as shown in Fig. 2.15. In processing a column of $\mathbf{r}$, a nonzero entry triggers the accumulation of a column of $\mathbf{\Phi}^T$ to compute $\mathbf{c}$. An array of N SAs is employed. The number of actual accumulations done by the SA array is $NVTS_r$. Since N SAs operate in parallel, the charge layer takes VTS_r to complete. Given the prototype specification, the charge layer takes

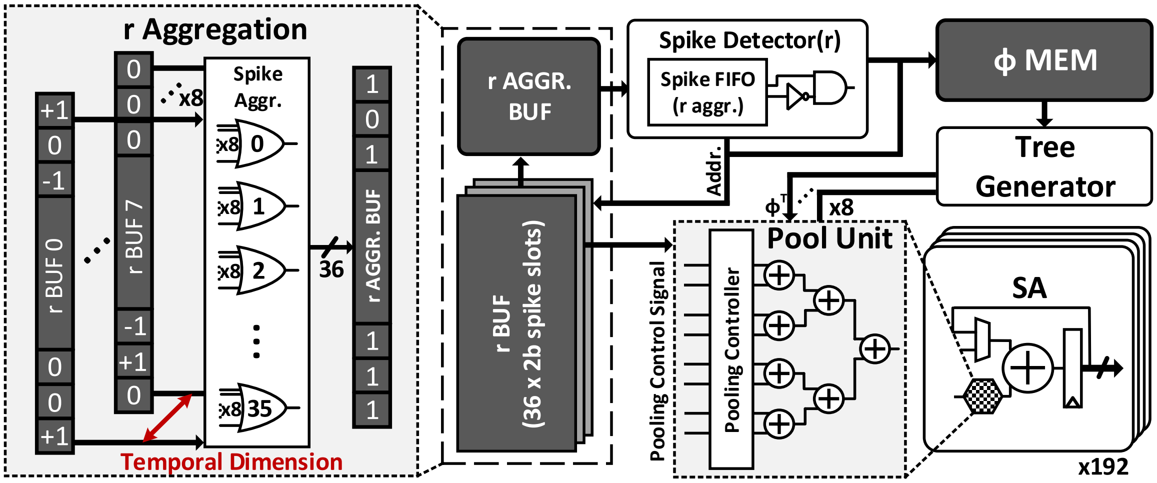


Figure 2.16: Charge layer design.

492 cycles.

To balance the layers, we apply temporal aggregation to shorten the latency of the charge layer. Each column of $\mathbf{r}$ represents a $X \times Y \times D$ frame. We compress $\mathbf{r}$ to $\mathbf{r}_a$ by pooling pixels at the same (x, y) location across D frames in a time series. If at least one of the D pixels is nonzero, pooling will output 1 for the pixel. After pooling, each entry of $\mathbf{r}_a$ represents an "aggregated" pixel i (in the XY-plane) across D frames. Note that temporal aggregation does not make use of any approximation. It essentially collects a vector of inputs and applies parallel processing. The technique has no impact on the encoding fidelity or classification accuracy.

Temporal aggregation enables shorter latency. As shown in Fig. 2.16, $\mathbf{r}_a$ is passed to a spike detector to output the first nonzero entry. As illustrated in Fig. 2.15, suppose the spike detector outputs address i (in the XY-plane), the address is used to read the D columns of $\mathbf{\Phi}^T$ that correspond to pixel i, and the D $\mathbf{r}$ values that are associated with pixel i. The D columns of $\mathbf{\Phi}^T$ are vector summed by the pool units located inside the SA array, as shown in Fig. 2.16, with the D $\mathbf{r}$ values as the control bits that determine whether the respective columns are zeroed, added or subtracted.

The aggregate processing increases the parallelism by a factor of D. The tem-

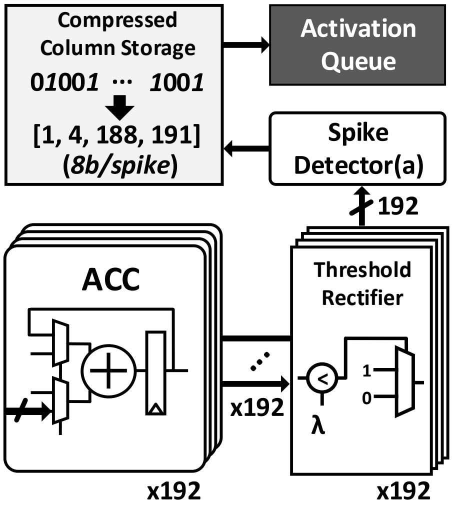


Figure 2.17: Activate layer design.

poral aggregation the D frames to one aggregate frame increases the density of 1's in the aggregate frame. If the D frames are completely independent, the density S'_r increases by D. However, the D frames belong to a time series and are highly correlated. In the prototype design, the density increases from 3% to 5%. With temporal aggregation and aggregate processing, the charge layer latency is reduced to $XYTS'_r$ $= 6\times6\times57\times5\% = 103$ cycles on average for the prototype design.

2.4.4 Activate Layer

The activate layer accumulates potential updates $\Delta\mathbf{u}$ ($N \times T$) to compute new neuron potentials. $\Delta\mathbf{u}$ is received column by column from the charge layer. The activate layer uses an array of N accumulators to update one column of potentials at a time. The potentials are thresholded to obtain binary activations $\mathbf{a}$.

The activations $\mathbf{a}$ ($N \times T$) are binary and sparse. As described in Section 2.4.2, $\mathbf{a}$ is fed to a spike detector to locate the nonzero entries for processing in the residual layer. The spike detector can be used to encode $\mathbf{a}$ in a CCS format, referring to storing only the addresses of nonzero entries in every column. as illustrated in Fig. 2.17.

Due to high sparsity, we can limit the number of nonzero entries in a column to

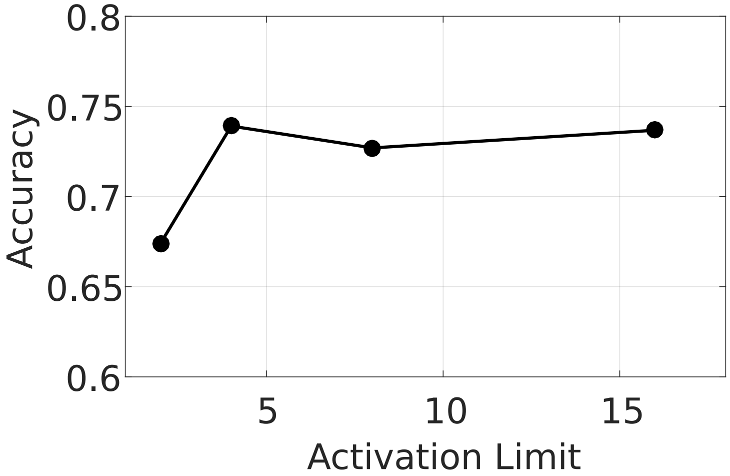


Figure 2.18: Effect of nonzero activation limit on classification accuracy.

a small fixed number. Simulations show that at least 4 nonzero activations need to be stored to ensure a high accuracy. If only 2 nonzero activations are stored, the accuracy is reduced by 10% as shown in Fig. 2.18. In the prototype design, we allow up to 8 nonzero activations to be stored. Additional nonzero entries are dropped with negligible impact on the accuracy due to the extremely low likelihood of occurrence. CCS effectively reduces the storage by 64%.

Putting the three layers together, the timing diagram for processing one 6×6×64 input is illustrated in Fig. 2.19. The input is divided into $T = 57$ temporally overlapped frames to be dispatched to the 3-layer processing in series. The processing is repeated for $I = 8$ iterations. Input data stream through the layers in sequence.

2.4.5 Summary of Design Optimizations

In the above subsections, we present the design techniques based on the prototype specification. The techniques are generally applicable and not limited to the given specification.

To quantify the benefits of the design techniques, we synthesized a baseline design in 40nm CMOS, along with design points after every step of the optimization. The

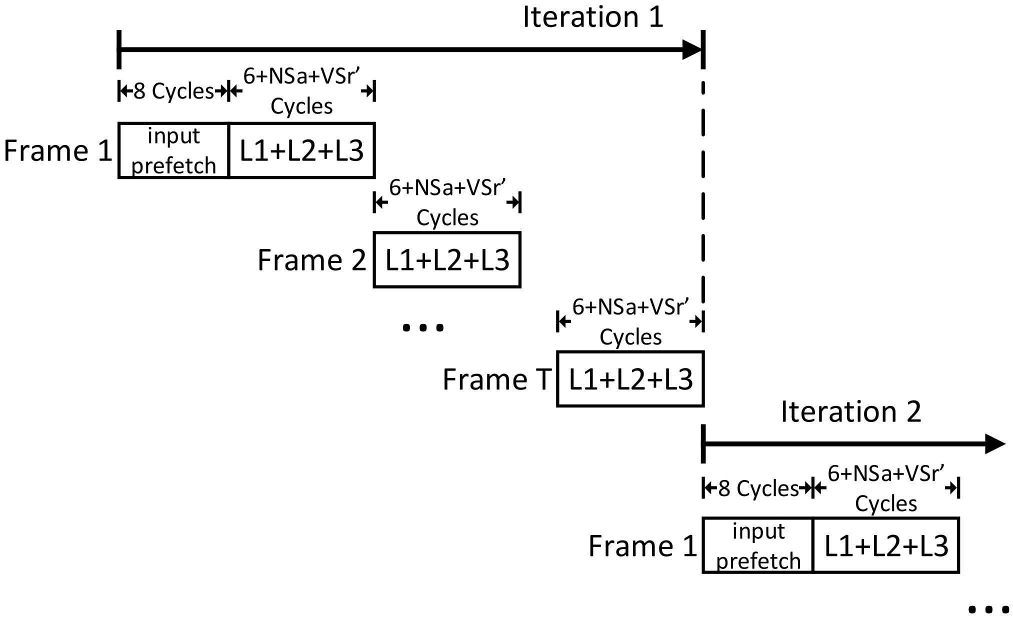
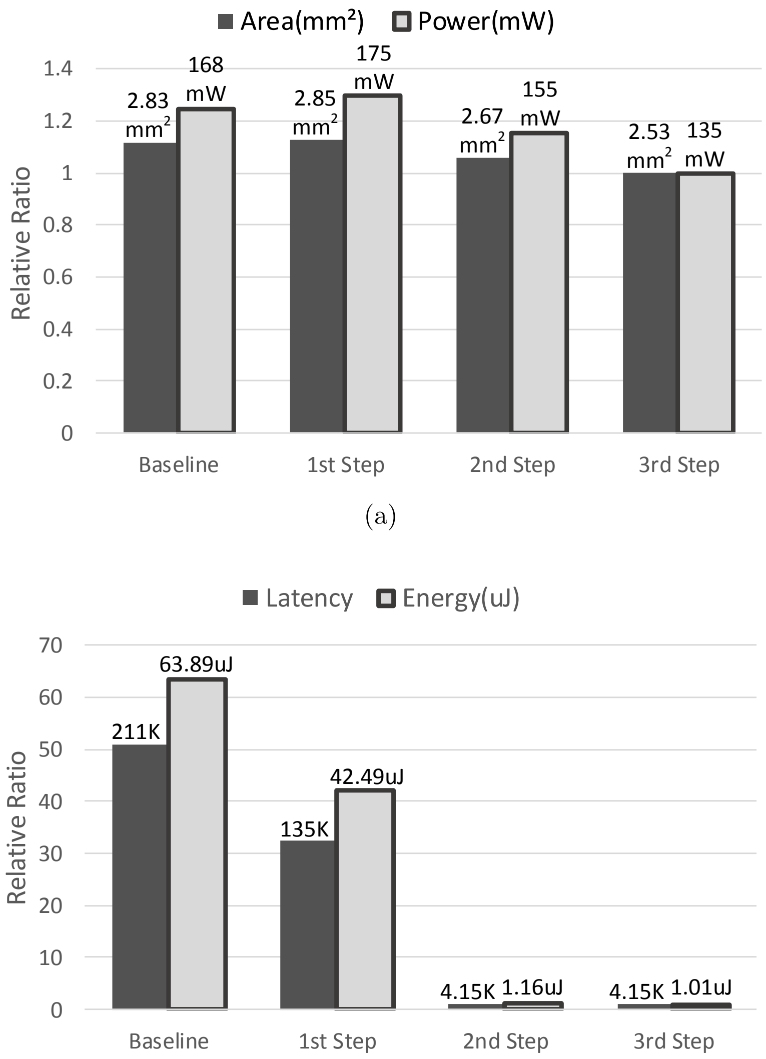


Figure 2.19: Timing illustration.

results are shown in Fig. 2.20. The baseline design employs a V-parallel MAC array in the residual layer, an N-parallel MAC array in the charge layer, and an N-parallel accumulator array in the activate layer. The design uses dense processing without spike detectors; and the residuals are not rectified. The baseline design reflects a standard parallel implementation without any sparsity or spiking optimizations. The latency of one iteration of processing is 211k cycles. The design is estimated to occupy 2.83mm^2 and consume 168mW.

The residual and charge layers account for the majority of the workload. Introducing sparsity optimizations has a major impact on the performance and the energy efficiency. In the first step of the optimization, we take advantage of sparse binary neuron activations to change the MAC array in the residual layer to an SA array, and use a spike detector to skip computations when activation is 0. The area and power increase by 1% and 4%, respectively, to support the net increase of the spike detection overhead minus the savings of the SA array, and the processing latency decreases by 36%. The latency is now entirely dominated by the charge layer.



(a)

(b)

Figure 2.20: (a) Area and power and (b) latency and energy after three design optimization steps: 1) sparse activation, 2) residual rectification, and 3) compressed activations.

In the second step, we apply ternary rectification to the residuals to change the MAC array to an SA array, and apply temporal aggregation to the charge layer. The area and power are reduced by 7% and 11%, respectively, and the latency is reduced by 32×.

In the third step, we compress the activations stored in the activate layer. The compression results in 5% area reduction and 13% power reduction.

In total, the three optimization steps increase the throughput by 51×, reduce the energy by 63×, and the area is reduced by 11%. Assume the KTH dataset with 6×6×64 inputs and the following parameter settings: $N = 192$, $X \times Y \times D = 6 \times 6 \times 8$, temporal stride of 1, spatial stride of 6, $I = 8$, $S_a = 1\%$ and $S_r = 3\%$. At a clock frequency of 240MHz, the real-time processing of 1080p HD video at 60 frames per second (FPS) requires the processing of a 6×6×64 input to be completed in 4.16k cycles. The optimizations proposed in this work are crucial for meeting this latency requirement.

Lastly, note that activation sparsity and ternary rectification of residuals caused most of the accuracy loss as shown in Fig. 2.5. However, these two techniques also contributed most of the performance and energy efficiency gain, as shown in Fig.2.20.

2.5 Prototype Implementation, Measured Results and Comparison

We design a prototype SoC for video inference applications. The system block diagram is shown in Fig. 2.21. The core of the SoC chip is the video inference processor that is made of three compute layers and memory to store dictionary, neuron potentials and input video frames for testing. The SoC also consists of an OpenRISC processor for programming, control, configuration and classification. Through the OpenRISC processor, the video inference processor is configurable with several

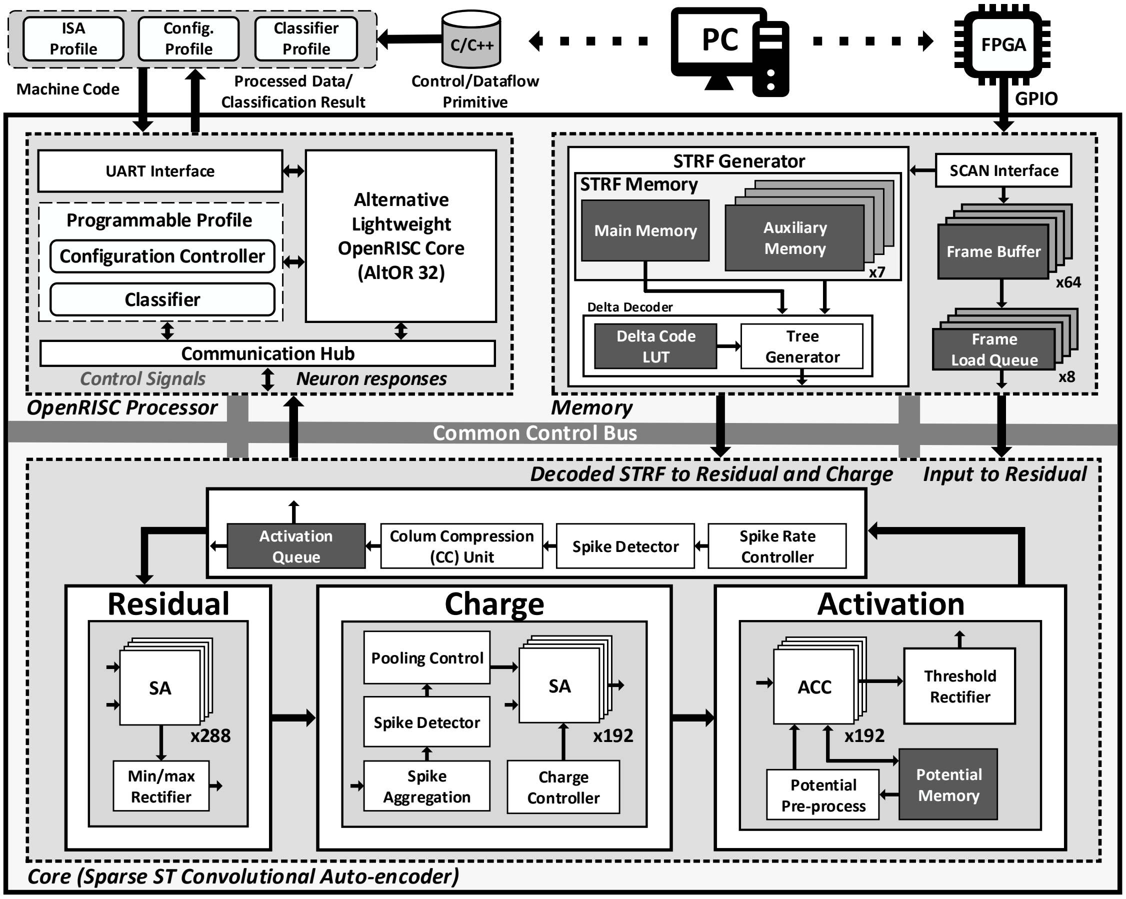


Figure 2.21: System-level design of video inference processor.

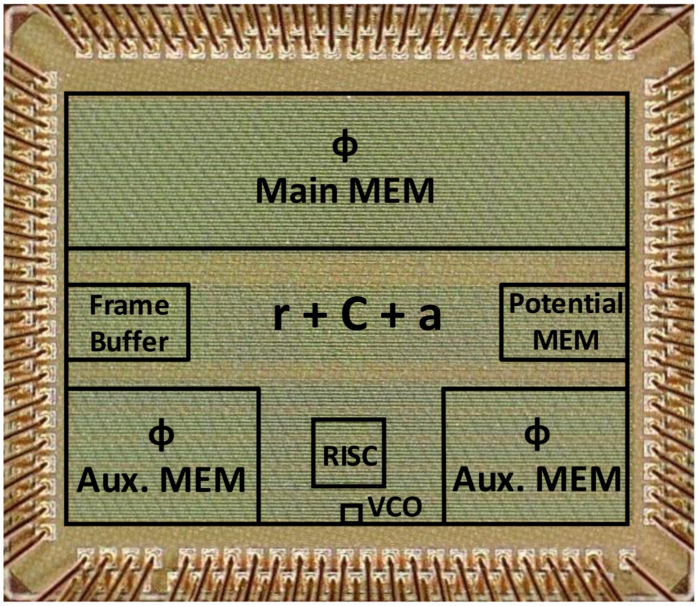


Figure 2.22: Microphotograph of the video inference SoC chip in 40nm CMOS.

Table 2.4: Action Classification Results of KTH Human Action Database

	Boxing	Clapping	Waving	Jogging	Running	Walking	Average
On-chip softmax classifier	70.0%	68.4%	85.0%	73.7%	94.4%	70.0%	76.7%
Off-chip SVM classifier	85.0%	78.9%	85.0%	73.7%	94.4%	80.0%	82.8%

settings: 64, 128 or 192 neurons (N), frame size ($X \times Y$) from 1 to 36 and depth (D) from 1 to 8.

The video inference SoC chip is implemented in 40nm CMOS, occupying 3.98mm^2. The core area measures 1.77mm × 1.43mm. The chip photo is shown in Fig. 2.22. The chip is tested for the KTH dataset with 6×6×64 inputs and the following parameter settings: N = 192, $X \times Y \times D$ = 6×6×8, temporal stride of 1, spatial stride of 6, I = 8, S_a = 1% and S_r = 3%. At room temperature, the chip is measured to achieve an effective performance of 1.70TOPS (including skipped OPs) at 0.9V and 240MHz. The performance meets the 60FPS 1920×1080 HD video data rate, while dissipating 135mW. The measured power and performance at room temperature are shown in Fig. 2.23.

The 6-class KTH Human Action Dataset [58] is used for action classification testing, with 600 samples and a training/testing split ratio of 5:1. Using the core extract-

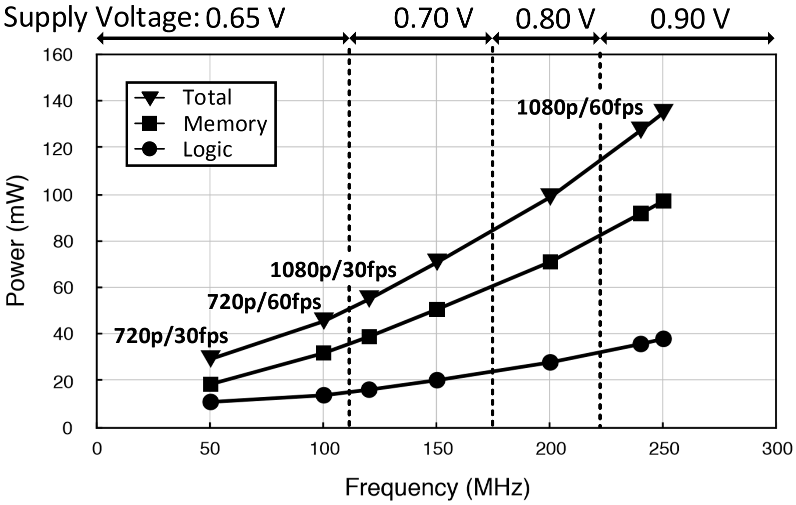


Figure 2.23: Measured power and performance of the video inference SoC chip.

ing the activation response of STRFs and a soft-max classifier programmed on the OpenRISC processor, the SoC achieves a 76.7% classification accuracy.

We also designed an SVM classifier based on a feed-forward network with two hidden layers of 40 and 50 neurons. The inputs to the classifier are extracted STRF features, i.e., spiking neuron outputs of the feature extraction network; and the outputs are the action class labels. The SVM is trained using a conjugate gradient method. The SVM classifier achieves an 82.8% accuracy as shown in Table 2.4.

In software and full precision, the state-of-the-art for the KTH dataset classification has now reached 92% accuracy [59]. The approach used differential gating of long short-term memory (LSTM), and the LSTM model consists of 450 input units, 300 memory cell state units, and 6 output units. There is not yet a clear path towards an efficient implementation of such a large model. In comparison, we sacrificed about 10% accuracy to obtain an efficient hardware implementation.

In Table 2.5, this work is compared with video processors for keypoint matching [42] in SIFT-based object recognition, and DPM-based object detection [44], as well as a convolutional sparse coding processor for feature and depth extraction [54]. Direct comparisons are not possible due to the major differences in algorithms and

Table 2.5: Comparison with Prior Work

	JSSC'15 Lee [42]	JSSC'17 Suleiman [44]	JSSC'18 Liu [54]	This Work
Application	Object matching	Object detection	Feature & depth extraction	Action classification
Algorithm	Vocabulary forest	Deformable parts model	LCA	LCA
Process	65nm	65nm	40nm	40nm
Core Area (mm^2)	2.3	12.8	2.56	2.53
Voltage (V)	1.2	0.77 - 1.11	0.6 - 0.9	0.65 - 0.9
Frequency (MHz)	250	62.5 - 125	120 - 380	50 - 250
Power (mW)	27.6	58.6 - 217	45 - 257	29.2 - 135
Performance (TOPS)	0.191	0.068 - 0.137	0.227 - 0.718 [a]	0.340 - 1.700 [b]
Power Efficiency (TOPS/W)	6.920	1.169 - 0.624	5.038 - 2.793	14.946 [c] - 12.583

[a] 1 OP is defined as an 8b MAC. [b] 1 OP is defined as an 8b add. (including skipped OPs)
[c] Power efficiency is 14.946TOPS/W at 0.65V, 100 MHz, (including skipped OPs)

applications. This work is the first video action classification processor that extracts spatio-temporal features from videos for sequence classification. The 2.53mm^2, 40nm test chip achieves up to 1.70TOPS at a power efficiency above 12.5TOPS/W (including skipped OPs). The performance and power efficiency are competitive with the other designs. Compared to [54] that used a similar algorithm for feature extraction and depth extraction, this work demonstrates higher performance and power efficiency.

2.6 Conclusion

We present an inference SoC for video sequence classification based on an RNN implementing LCA, a neuro-inspired compressed sensing algorithm. Due to the large video data size, spatio-temporal, and iterative processing, the computational requirement of the RNN is high. We adopt a residual form of the LCA algorithm and apply a transformation by rectifying the residuals after each inference iteration to ternary spikes.

The algorithm reformulation leads to a sparse all-spiking RNN architecture realized in three layers: residual layer, charge layer and activate layer. All layers are implemented primarily in SAs. Data are seamlessly streamed across the layers in iterations. To balance the processing layers and avoid stalling, we use a temporal aggregation and aggregate processing technique to shorten the processing latency of the slowest charge layer. To reduce the chip area and power, we apply delta compression and non-uniform quantization to STRFs to reduce the memory by 42% and CCS encoding to sparse activations to reduce the memory by 64%. In all, the algorithm and architecture techniques increase the processing throughput and reduce the energy by 51× and 63×, respectively, while the area is kept nearly constant.

The design is prototyped in a 2.53mm^2 40nm CMOS video inference SoC chip. The chip is measured to achieve 1.70TOPS (including skipped OPs) at 0.9V and 250MHz, dissipating 135mW. Tested with the 6-class KTH Human Action Dataset, the chip provides a 76.7% classification accuracy.

Not every video application in practice can directly benefit from a design that supports $XY \leq 36$. As a small research prototype, we chose $XY = 36$ to target the KTH dataset. Even for this relatively small dataset, multiple optimizations are needed to keep the hardware complexity within bounds. Video sequence classification is a demanding task. For larger and practical applications, we expect more substantial compute resources to be needed. The same optimizations demonstrated in this work are equally applicable to larger and more demanding applications.

CHAPTER III

Design of SRAM-Based Accelerator for Solving Partial Differential Equations

3.1 Introduction

Many physical phenomena, such as heat and fluid dynamics, are described by partial differential equations (PDE). Most PDEs are solved numerically, by first quantizing the solution space in a grid and then applying iterative methods to refine the solution to a desired error tolerance [60].

High-precision PDE solutions require fine grids and high numerical precision, leading to a significant amount of data that needs to be processed, moved and stored. Moreover, a PDE solver commonly requires tens of thousands of iterations to converge. For example, solving a 2D Poisson equation using a 128×128 grid on a graphics processing unit (GPU) in floating-point is estimated to take 15mJ/iteration. To shrink the error tolerance from 10^{-4} to 10^{-7} costs at least 320J!

High-performance digital PDE solvers have been proposed [61, 62], but they still require high-bandwidth DRAM access to sustain the massive number of parallel, high-precision compute. Analog computers [63,64] were proposed to accelerate PDE solvers by approximate compute to reduce the IO requirement and mixed-signal approach to speed up core computations. However, analog compute requires large area, limiting the parallelism and efficiency in supporting large PDE problems.

To enable a more efficient and practical PDE accelerator design, we apply two algorithm approaches: 1) we adopt a multigrid method that divides the PDE solver to a fine-grid compute and a coarse-grid compute and iterates between the two to accelerator convergence by 5 to 10× on average; 2) we transform both fine-grid compute and coarse-grid compute to a residual form, lowering the precision to 5b for a low error tolerance below 10^{-8}.

Even with faster convergence and much-reduced precision, the implementation cost can still be high using a conventional digital approach. Recently, process in memory (PIM) has been proposed as a new technique that computes directly on a large array of data in place, within memory [27, 65]. SRAM-based PIM relies on level- and/or width-modulating word lines (WL) of the SRAM array to encode multipliers, and activating multiple WLs in parallel [27, 30, 65, 66]. The SRAM cells' currents in discharging the bit lines (BL) represent the products, and the current on each BL represents the sum of products. Alternatively, BLs can be modulated to encode multipliers, and BLs are joined to produce the sum of products [29]. By partly eliminating data movement cost and providing a high degree of parallelism, PIM holds the potential of achieving both high performance and efficiency in tasks that involve parallel multiply-accumulate (MAC) operations, such as classification and neural networks. Prior work has demonstrated PIM in SRAM that achieved 633.4 pJ/classification [27] and 1.2 nJ/classification [66], translating to 1.17 pJ/op where an op is defined as a multiply-accumulate (MAC) operation.

Current SRAM-based PIM designs are limited by SRAM's binary storage and the overhead of multi-bit A/D conversion. Some designs support only binary multiplicands [27, 29, 30]; and some choose binary outputs [27, 30]. To reduce the number of ADCs, some designs are tailored to computations in a cone structure that requires only one or a small number of ADCs at the final output [65, 66]. These approaches are not applicable to a PDE solver that requires iterative multi-bit operations and solutions to be updated in every iteration.

In this work, we combine the multigrid, residual algorithmic approach with the optimal mapping on 5b MAC SRAMs to produce a high-performance and efficient PDE solver accelerator. A MAC SRAM supports 5b×5b MACs with full-bandwidth 5b outputs to support a PDE solver. We design a DLL-based 5b driver that produces WL pulses down to $\frac{1}{8}$ of a clock period with PVT tolerance. The WL pulses are level-modulated to match 5b multiplicands stored in SRAM. We employ a dual-ramp single-slope ADC [67] that employs a coarse ramp followed by a fine ramp to increase conversion speed and minimize area. A 1.87mm^2 180nm chip consisting of four 320×64 MAC SRAMs is demonstrated at 200MHz, each providing 1.42G MAC/s and 32 5b ADCs at a power consumption of 16.6mW.

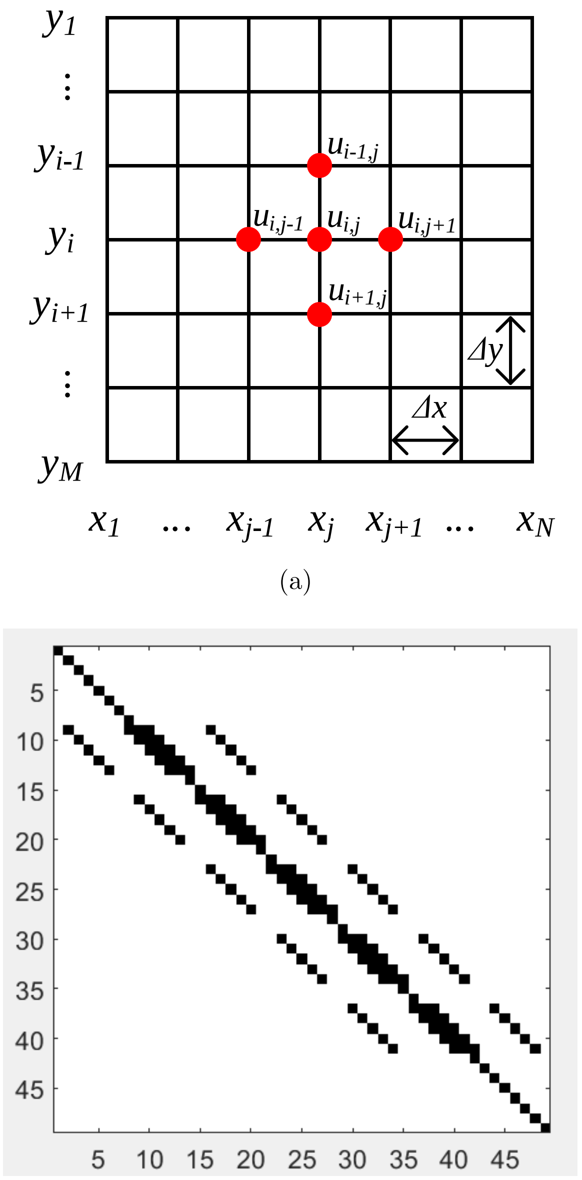
3.2 Numerical PDE Solver by Finite Difference Method and Jacobi Iteration

We use the solution to 2D Poisson's equation, shown in (3.1), to explain the PDE solver design. Poisson's equation is widely used in practical applications [68].

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = b, \tag{3.1}$$

where $b(x, y)$ is given, boundary conditions are specified on the perimeter of the domain, and the solution $u(x, y)$ is sought.

Most PDE problems do not have analytical solutions. Instead, numerical approaches using grid discretization is popular. For the 2D Poisson's equation above, the finite difference method (FDM) can be applied to convert u and b into a $M \times N$ grid of step size Δx and Δy along x and y as shown in Fig. 3.1(a) [38]. The discretization results in a system of MN equations:



(a)

(b)

Figure 3.1: (a) Illustration of a 2D finite difference grid; and (b) rendition of matrix **A** for a 7×7 grid.

$$\frac{u_{i-1,j}+u_{i+1,j}-2u_{i,j}}{\Delta x^2}+\frac{u_{i,j-1}+u_{i,j+1}-2u_{i,j}}{\Delta y^2}=b_{i,j}, \qquad (3.2)$$
$$2 \leq i \leq M-1, 2 \leq j \leq N-1,$$

where $u_{i-1,j}$ is the value of u at grid position $(i-1,j)$, $u_{i+1,j}$ is the value of u at $(i+1,j)$, and so on. $u_{x,y}$ is known on the boundaries of the grid, and unknown in the interior of the grid. The u values can be put in a $MN \times 1$ vector $\mathbf{u}$, and similarly the b values are put in a $NM \times 1$ vector $\mathbf{b}$. The system of equations (3.2) can be written as

$$\mathbf{Au}=\mathbf{b}, \qquad (3.3)$$

where $\mathbf{A}$ is a $MN \times MN$ matrix that stores the weights of u's in (3.2). For a sufficiently large grid, $\mathbf{A}$ is highly sparse. A rendition of matrix $\mathbf{A}$ for a 7×7 grid is shown in Fig. 3.1(b), where black dots indicate nonzero entries.

To solve u, the system of equations (3.2) can be rewritten by shifting the $u_{i,j}$ terms to the left:

$$\begin{aligned} u_{i,j} =& \frac{\Delta x^2}{2(\Delta x^2+\Delta y^2)}\left(u_{i-1,j}+u_{i+1,j}\right) \\ &+\frac{\Delta y^2}{2(\Delta x^2+\Delta y^2)}\left(u_{i,j-1}+u_{i,j+1}\right) \\ &-\frac{\Delta x^2 \Delta y^2}{2(\Delta x^2+\Delta y^2)}b_{i,j}, \\ =& s_1\left(u_{i-1,j}+u_{i+1,j}\right)+s_2\left(u_{i,j-1}+u_{i,j+1}\right)+c_{i,j}, \\ & 2 \leq i \leq M-1, 2 \leq j \leq N-1, \end{aligned} \qquad (3.4)$$

Starting from the boundary conditions and initial guess of interior points of u, the system of equations (3.4) can be solved iteratively by the Jacobi method:

$$
\begin{aligned}
u_{i,j}^{(n+1)} =& \frac{\Delta x^2}{2(\Delta x^2 + \Delta y^2)} \left(u_{i-1,j}^{(n)} + u_{i+1,j}^{(n)} \right) \\
&+ \frac{\Delta y^2}{2(\Delta x^2 + \Delta y^2)} \left(u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)} \right) \\
&- \frac{\Delta x^2 \Delta y^2}{2(\Delta x^2 + \Delta y^2)} b_{i,j}, \\
=& s_1 \left(u_{i-1,j}^{(n)} + u_{i+1,j}^{(n)} \right) + s_2 \left(u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)} \right) + c_{i,j}, \\
& 2 \le i \le M-1, 2 \le j \le N-1,
\end{aligned}
\tag{3.5}
$$

where the superscript (n) in $u_{i,j}^{(n)}$ indicates the value of $u_{i,j}$ in iteration n. Since Δx and Δy are known and $b_{i,j}$ does not change each iteration, s_1, s_2 and $c_{i,j}$ can be pre-computed. Each Jacobi iteration updates all $(M-2)(N-2)$ interior u values based on their values in the previous iteration. Using FDM, each $u_{i,j}$ update requires its neighboring 4 points, $\{u_{i-1,j}, u_{i+1,j}, u_{i,j-1}, u_{i,j+1}\}$, called a 4-point stencil, as illustrated in Fig. 3.1(a). In the matrix form, the Jacobi iterations can be described as

$$
\mathbf{u}^{(n+1)} = \mathbf{T}\mathbf{u}^{(n)} + \mathbf{c}, \tag{3.6}
$$

where $\mathbf{T}$ is a $MN \times MN$ matrix that stores the stencil weights. $\mathbf{T}$ essentially contains the off-diagonal entries of $\mathbf{A}$, and it is also highly sparse for a sufficiently large grid. Higher-order PDEs use higher-order grids, but the same formulation applies.

In addition to FDM, there are two other popular discretization methods: the finite element method (FEM) and the finite volume method (FVM) [69]. In FEM, a set of bases is defined to approximate integrals and allow discretization of any shape. The discretization is non-uniform. In FVM, each discretized point is viewed as a small sub-region, where the estimation is based on the flux from the entire sub-region surfaces. The stencil weights are non-sharable. Compared to FEM and FVM, FDM uses a uniform grid and common stencil weights, which enables highly parallel, regular computation with smaller memory. FDM is more computationally friendly than FEM or FVM, and amenable to acceleration.

3.3 Algorithm Formulation For Faster Convergence and Lower Precision

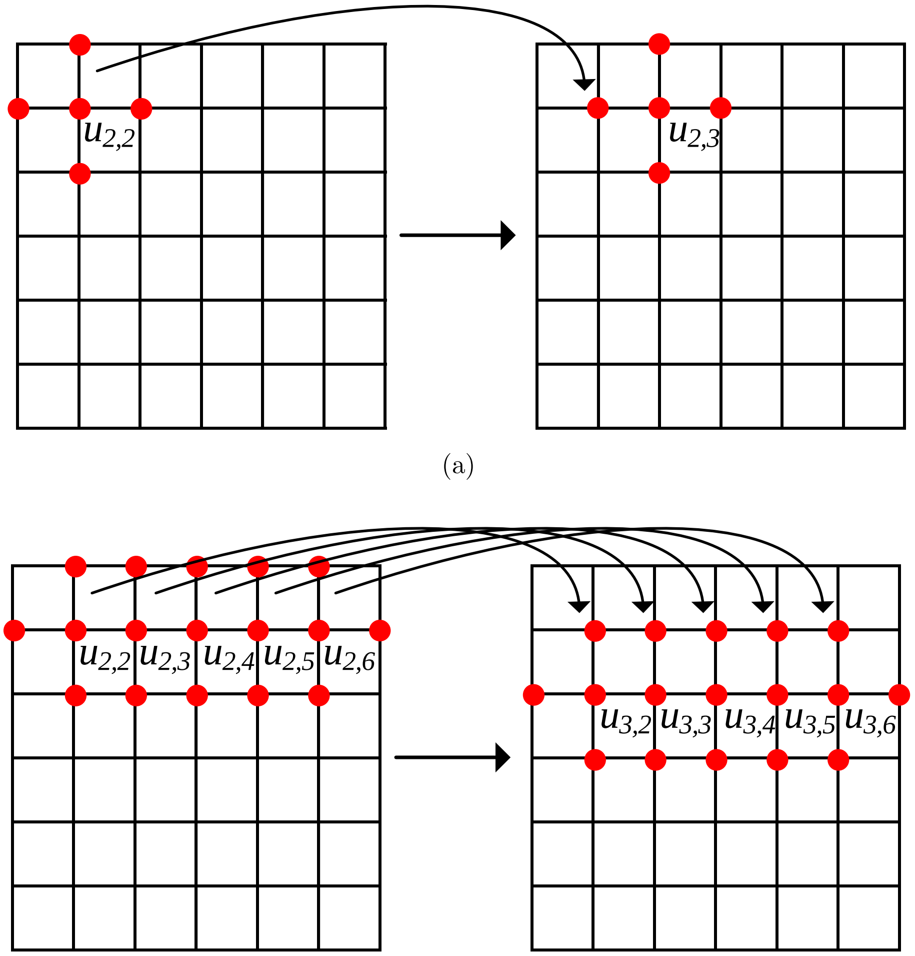
Our baseline PDE solver applies a single grid with the Jacobi method. To reach an accurate solution, a fine grid of fine step sizes is required. However, a fine grid results in a large volume of data and the Jacobi method has a relatively slow convergence rate. To speed up convergence, we adopt an alternative iterative method and a multigrid approach. The multigrid approach can also be transformed to the residual form to enable aggressive precision reduction to simplify the computation.

3.3.1 Hybrid Layer Update Method

Using the Jacobi method, the next iteration u values are computed based entirely on the u values from the current iteration. Thus, the next iteration cannot start until the current iteration is complete and saved in a buffer memory. To reach a faster convergence and possibly remove the buffer, the Gauss-Seidel method computes the next iteration u values based on the latest available u values.

Shown in Figure 3.2(a), the Gauss-Seidel method updates $u_{i,j}$ one by one, e.g., from left to right, and then top to bottom. The updated values are immediately applied in computing the neighboring $u_{i,j}$ value. By always using the latest available values, the Gauss-Seidel method leads to a faster convergence than the Jacobi method. The Gauss-Seidel method also removes the buffer, but it introduces data dependency: $u_{i,j}$ has to be updated in a sequential order, limiting parallelism.

The choice of iterative method trades parallelism with number of iterations. Although the Gauss-Seidel method takes fewer iterations, the Jacobi method with full


$u_{2,2}$
$u_{2,3}$
$u_{2,2}$ $u_{2,3}$ $u_{2,4}$ $u_{2,5}$ $u_{2,6}$
$u_{3,2}$ $u_{3,3}$ $u_{3,4}$ $u_{3,5}$ $u_{3,6}$

(a)

(b)

Figure 3.2: PDE iterative methods with sequential updates: (a) Gauss-Seidel update; (b) hybrid layer update.

parallelism computes quadratically faster per iteration, which dominates the overall computational speed. However, for a large PDE problem, limited computational resource and memory storage efficiency often prohibit fully parallel iteration updates. To speed up convergence given the available parallelism, we employ a hybrid layer update method [70, 71].

Shown in Figure 3.2(b), the grid is divided into layers. A layer of u values can be updated in parallel without data dependency, following the Jacobi method. Updated u values from one layer are used in computing the updates for the next layer, following the Gauss-Seidel method. For a large grid, plenty of parallelism is available in one layer to enable parallel processing. The layer-to-layer sequential update using Gauss-Seidel provides a faster convergence. The hybrid layer update method uses a layer buffer, smaller than a block buffer needed for the complete Jacobi method.

3.3.2 Multigrid and Low-Precision Complete Residual Approach

To further speed up convergence, the multigrid method introduces a $m \times n$ ($m < M$, $n < N$) coarse grid in addition to the fine grid [38]. Coarse grid vertices represent a local region of grid values. By interleaving coarse-grid iterations with fine-grid iterations, convergence is accelerated thanks to faster propagation. A coarse grid reduces the computation workload, e.g., using a 2×2 downsampled coarse grid reduces the workload by 75%.

Illustrated in Figure 3.3(a), the residual $\mathbf{r}$ is obtained after a round of fine-grid iterations, and restricted to $\mathbf{r}^*$ by multiplying by a $mn \times MN$ restriction matrix $\mathbf{R}$. After transitioning to the coarse grid, we can use an iteration method, e.g., the layer update method, to obtain $\mathbf{e}^*$ in $\mathbf{A}^*\mathbf{e}^* = \mathbf{r}^*$. Note that the $mn \times mn$ matrix $\mathbf{A}^*$ is the downsampled version of $\mathbf{A}$. After a round of coarse-grid iterations, $\mathbf{e}^*$ is interpolated by multiplying by a $MN \times mn$ interpolation matrix $\mathbf{I}$, and then it is used to update $\mathbf{u}$ to start the next round of fine-grid iterations. The restriction and interpolation are

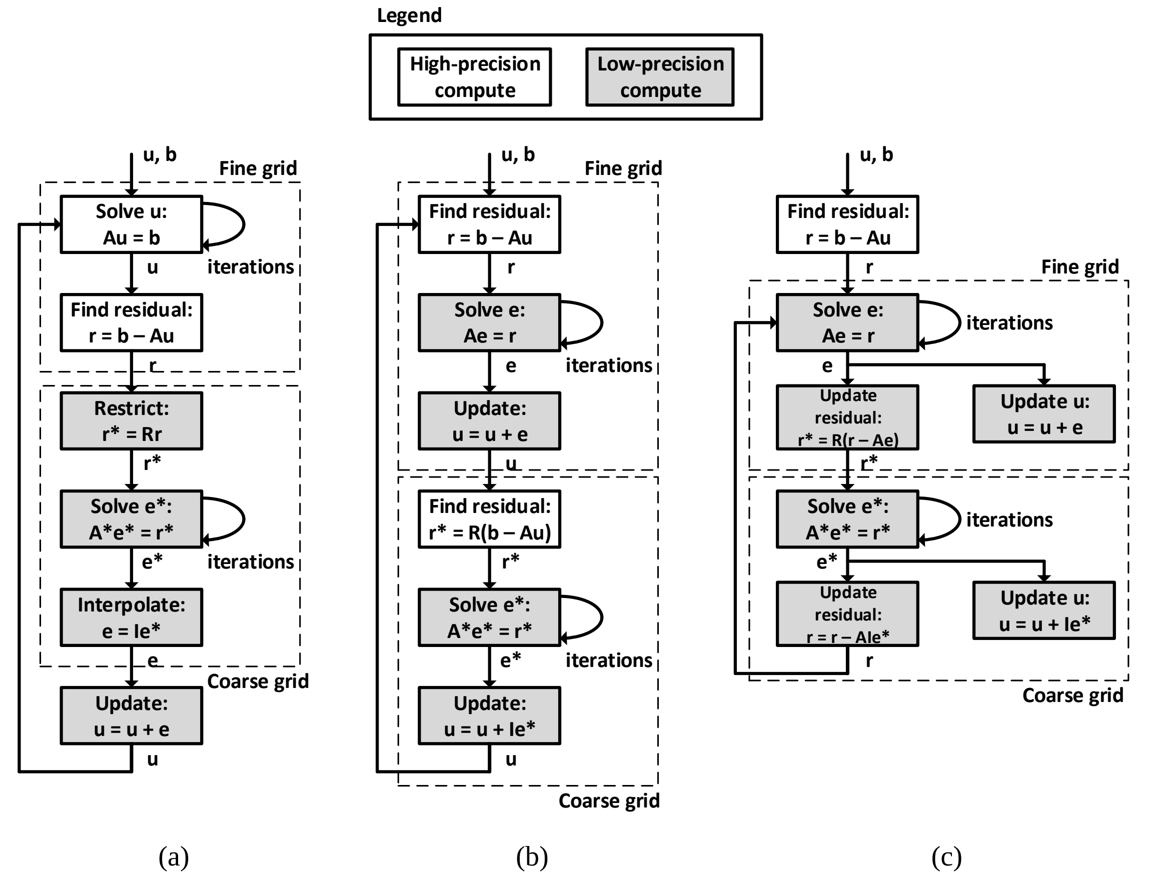


Figure 3.3: Residual approaches: (a) standard residual approach applied to fine grid only; (b) complete residual approach applied to both fine grid and coarse grid; and (c) reformulated complete residual approach without high-precision multiplication.

commonly done by pooling and averaging.

Because coarse-grid compute operates on errors $\mathbf{e}^*$ and residuals $\mathbf{r}^*$, i.e., the small differences between consecutive iterations, the precision of the errors and the residuals can be relaxed. Realizing the potential benefit of the residual approach, we extend the residual approach to fine-grid compute, as shown in Figure 3.3(b), so that the fine-grid iterations also work on errors $\mathbf{e}$ and residuals $\mathbf{r}$.

Note that in Figure 3.3(b), after a round of fine-grid or coarse-grid iterations, $\mathbf{u}$ is updated, and then the updated $\mathbf{u}$ is used to update the residuals. Since $\mathbf{u}$ is in full precision, updating the residuals requires costly multiplications $\mathbf{Au}$. To avoid

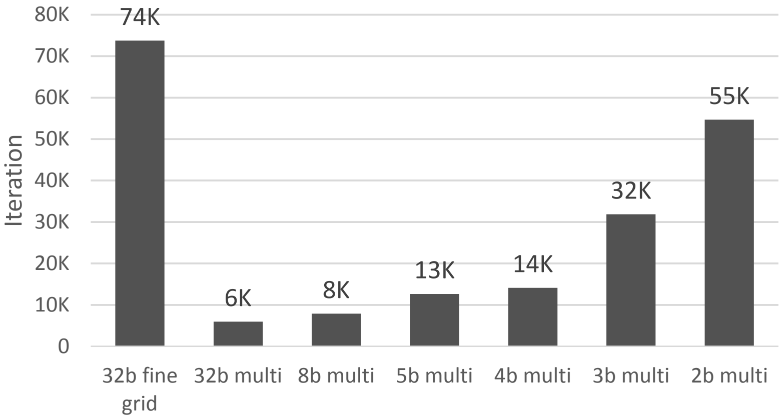


Figure 3.4: Latency of solving 2D Poisson's equation to reach an error tolerance of 10^{-7} using a 127×127 single-grid and 32b, 8b, 5b, 4b, 3b, and 2b quantized multigrids (a 127×127 fine grid combined with a 64×64 coarse grid).

full-precision multiplication, we note that $\mathbf{r}^{(i+1)} = \mathbf{b} - \mathbf{A}\mathbf{u}^{(i+1)} = \mathbf{b} - \mathbf{A}(\mathbf{u}^{(i)} + \mathbf{e}^{(i)}) = \mathbf{r}^{(i)} - \mathbf{A}\mathbf{e}^{(i)}$, where the superscripts indicate iteration number. Using this identity, the complete residual approach is reformulated to Figure 3.3(c). We dropped the iteration number in the figure for simplicity. Note that the reformulation does not require any high-precision multiplications.

3.3.3 Evaluation of Algorithm Improvement

In Figure 3.4, we compare the average convergence speed of solving 2D Poisson's equation using the Jacobi method and an error tolerance 10^{-7}. A 32b-quantized 127×127 grid is applied using the baseline approach (single grid, non-residual formulation), and 74k iterations are needed to reach convergence. If the single grid is replaced by multi grids, i.e., a 127×127 grid and a 64×64 grid, the convergence is shortened by more than 12× to 6k iterations.

The complete residual approach allows the precision to be aggressively reduced to allow a shorter bit width, but at the cost of slower convergence. For example, when the computations are quantized to 8b, the latency increases by 33%. Further quantizing to 5b and 4b results in 2.1× and 2.3× latency increase, respectively. A shorter bit

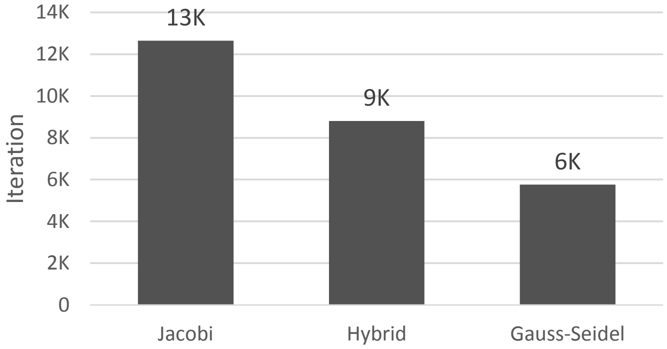


Figure 3.5: Latency of solving 2D Poisson's equation with 5b quantization to reach an error tolerance of 10^{-7} using 5b multigrid and Jacobi, hybrid layer update, and Gauss-Seidel methods.

width is preferred for lower implementation cost, and the optimal is between 4b and 8b. Reducing the bit width below 4b slows down the convergence drastically, and ceases to be practical.

In Figure 3.5, we compare the convergence speed of Jacobi, hybrid layer update, and Gauss-Seidel methods, assuming 5b computations and the same experimental setup as above. The layer update method results in 31% fewer iterations compared to the Jacobi method. Although the layer update converges 50% slower than the Gauss-Seidel method, it provides the opportunity for more parallel processing to achieve a higher throughput.

3.4 Mapping of PDE Solver on MAC SRAM

The core computation in a multigrid PDE solver is for solving $\mathbf{Ae} = \mathbf{r}$ in the fine grid, and $\mathbf{A}^*\mathbf{e}^* = \mathbf{r}^*$ in the coarse grid, as shown in Figure 3.3(c). Similar to solving $\mathbf{Au} = \mathbf{b}$, both $\mathbf{Ae} = \mathbf{r}$ and $\mathbf{A}^*\mathbf{e}^* = \mathbf{r}^*$ are solved by iterations:

$$\begin{aligned} \mathbf{e}^{(n+1)} &= \quad \mathbf{Te}^{(n)} + \mathbf{c} \\ \mathbf{e}^{*(n+1)} &= \mathbf{T}^*\mathbf{e}^{*(n)} + \mathbf{c}^*, \end{aligned} \tag{3.7}$$

where $\mathbf{e}^{(n+1)}$ and $\mathbf{e}^{(n)}$ are $MN \times 1$ vectors, $\mathbf{T}$ is a $MN \times MN$ matrix, $\mathbf{e}^{*(n+1)}$ and $\mathbf{e}^{*(n)}$ are $mn \times 1$ vectors, and $\mathbf{T}^*$ is a $mn \times mn$ matrix. The matrix-vector products, $\mathbf{T}\mathbf{e}^{(n)}$ and $\mathbf{T}^*\mathbf{e}^{*(n)}$, take the most computation resources.

We adopt the complete residual approach to quantize the computation to 5b, making it possible to perform the computation using MAC SRAM. Since the stencil matrices $\mathbf{T}$ and $\mathbf{T}^*$ are highly sparse, it is wasteful to be stored in SRAM. Instead, we store the errors $\mathbf{e}$ and $\mathbf{e}^*$ in SRAM with each 5b value stored in 5 cells in consecutive rows, and 5b stencil weights are applied as WL pulses to the SRAM. The MAC outputs are the updated errors, which are converted to 5b digital values.

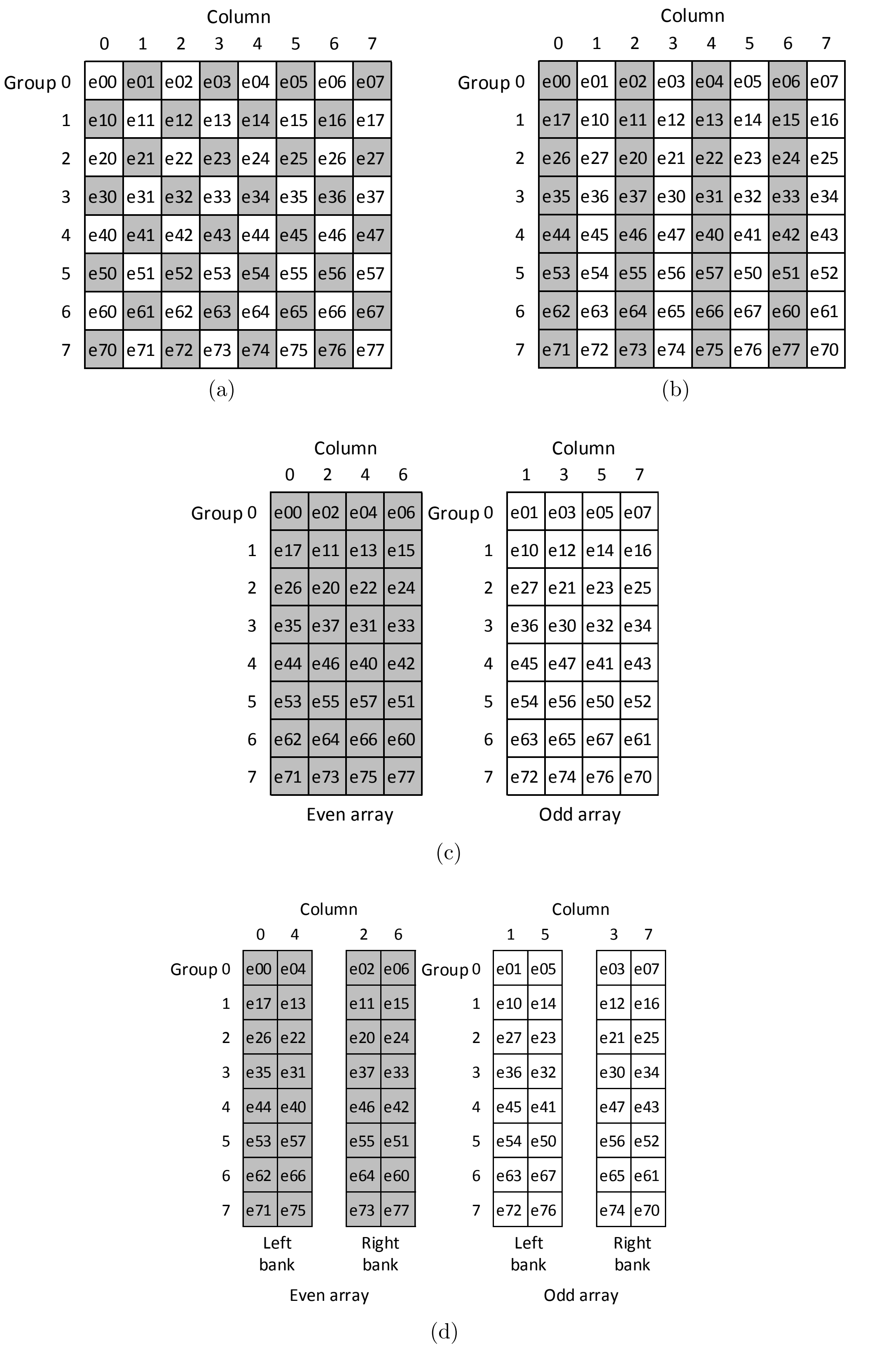
3.4.1 Direct Mapping

In the direct mapping, a $5M \times N$ SRAM array can be used for the $M \times N$ grid, and the $\mathbf{e}$ or $\mathbf{e}^*$ values are stored in the SRAM based on their grid locations. The mapping is illustrated in Figure 3.6(a). Note that a row in Figure 3.6(a) represents 5 consecutive rows in memory as each 5b operand spans 5 rows. To avoid confusion, we will use "group" to refer to a row of 5b operands.

As an example, to update $e_{1,1}$, $\{e_{1,0}, e_{1,2}, e_{0,1}, e_{2,1}\}$ need to be read, multiplied by their respective stencil weights and then the partial sums are added. However, the four operands $\{e_{1,0}, e_{1,2}, e_{0,1}, e_{2,1}\}$ are not located on the same WL or BL, thus it is impossible to add the partial sums in PIM. The direct mapping is also incompatible with the hybrid layer update method due to BL access conflicts that prevent $\mathbf{e}$ values to be computed in parallel.

3.4.2 Rotation Mapping

To use PIM, the operands need to be aligned in memory. We create a rotation mapping to transform Figure 3.6(a) to Figure 3.6(b): group 0 stays in place, group 1 is right rotated by 1, and group i is right rotated by i, and so on. The rotation



	Column							
	0	1	2	3	4	5	6	7
Group 0	e00	e01	e02	e03	e04	e05	e06	e07
1	e10	e11	e12	e13	e14	e15	e16	e17
2	e20	e21	e22	e23	e24	e25	e26	e27
3	e30	e31	e32	e33	e34	e35	e36	e37
4	e40	e41	e42	e43	e44	e45	e46	e47
5	e50	e51	e52	e53	e54	e55	e56	e57
6	e60	e61	e62	e63	e64	e65	e66	e67
7	e70	e71	e72	e73	e74	e75	e76	e77

(a)

	Column							
	0	1	2	3	4	5	6	7
Group 0	e00	e01	e02	e03	e04	e05	e06	e07
1	e17	e10	e11	e12	e13	e14	e15	e16
2	e26	e27	e20	e21	e22	e23	e24	e25
3	e35	e36	e37	e30	e31	e32	e33	e34
4	e44	e45	e46	e47	e40	e41	e42	e43
5	e53	e54	e55	e56	e57	e50	e51	e52
6	e62	e63	e64	e65	e66	e67	e60	e61
7	e71	e72	e73	e74	e75	e76	e77	e70

(b)

	Column			
	0	2	4	6
Group 0	e00	e02	e04	e06
1	e17	e11	e13	e15
2	e26	e20	e22	e24
3	e35	e37	e31	e33
4	e44	e46	e40	e42
5	e53	e55	e57	e51
6	e62	e64	e66	e60
7	e71	e73	e75	e77

Even array

	Column			
	1	3	5	7
Group 0	e01	e03	e05	e07
1	e10	e12	e14	e16
2	e27	e21	e23	e25
3	e36	e30	e32	e34
4	e45	e47	e41	e43
5	e54	e56	e50	e52
6	e63	e65	e67	e61
7	e72	e74	e76	e70

Odd array

(c)

	Column			
	0	4	2	6
Group 0	e00	e04	e02	e06
1	e17	e13	e11	e15
2	e26	e22	e20	e24
3	e35	e31	e37	e33
4	e44	e40	e46	e42
5	e53	e57	e55	e51
6	e62	e66	e64	e60
7	e71	e75	e73	e77
	Left bank		Right bank	

Even array

	Column			
	1	5	3	7
Group 0	e01	e05	e03	e07
1	e10	e14	e12	e16
2	e27	e23	e21	e25
3	e36	e32	e30	e34
4	e45	e41	e47	e43
5	e54	e50	e56	e52
6	e63	e67	e65	e61
7	e72	e76	e74	e70
	Left bank		Right bank	

Odd array

(d)

Figure 3.6: PDE mapping on MAC SRAM: (a) direct mapping; (b) rotation mapping; (c) array splitting, and (d) bank splitting.

allows the operands to be aligned. For example, activating group 0 and group 1 with the respective stencil weights enables the parallel summing of pairs of partial sums for updating $e_{1,i}, 2 \leq i \leq M-1$.

With rotation mapping, the operands from the odd-numbered columns are read and the results are written to the even-numbered columns, and vice versa. Therefore we split the odd and even columns and store them in separate even and odd SRAM arrays as shown in Figure 3.6(c). Each array provides two ports: one port for read to perform MAC operations, and another port for write-back. The odd and even arrays run in parallel. The read output of the odd array is written back to the even array, and vice versa.

After the rotation mapping, one 4-point stencil is separated into two halves. For example, the stencil $\{e_{1,0}, e_{1,2}, e_{0,1}, e_{2,1}\}$ stored in the odd SRAM array needs to be separated to two banks: one that stores $\{e_{1,0}, e_{0,1}\}$ and the other that stores $\{e_{1,2}, e_{2,1}\}$ as shown in Figure 3.6(c). The partial sums of the two halves are summed on two BLs after activating group 0 and group 1 for the left bank, and group 1 and group 2 for the right bank. To resolve this lane misalignment, each SRAM array is further split to the left and right bank, as shown in Figure 3.6(c).

To sum up, a $M \times N$ grid is stored in two SRAM arrays, each consisting of two banks of size $5M \times \frac{N}{4}$. The complete update of a layer requires two steps across the two SRAM arrays. For example, the update of layer 0 is done in two steps across the two SRAM arrays: 1) group 0 and group 1 are activated on the left bank, while group 1 and group 2 are activated on the right bank. The respective BLs from the two banks are joined to complete the summing; and 2) group 1 and group 2 are activated on the left bank, while group 0 and group 1 are activated on the right bank. The respective BLs are then joined to complete the summing.

The rotation mapping and the array and bank splitting offer a number of advantages: 1) the memory is nearly fully utilized to support stencil computation without duplicate storage; 2) simple and regular BL muxing and control; and 3) compatible with the hybrid layer update method. One possible drawback of the approach is the lower array efficiency due to the splitting into smaller SRAM arrays and banks. However, for a sufficiently large grid size that is common in the most demanding applications, the loss of efficiency due to banking is minimized.

We also note that in a typical PIM design, all rows of the memory are activated at the same time to unleash the full parallelism [27, 30], but it is at the cost of reduced BL precision. Our approach activates a subset of rows of the memory, i.e., 10 rows, sacrifices the performance, but it also reduces the BL precision and simplifies the ADC design.

3.5 Prototype Architecture

A prototype PDE solver chip is designed in 180nm CMOS. The CMOS chip implementation and testing was done in collaboration with Jacob Botimer. The chip architecture is shown in Figure 3.7. It consists of an iteration module, a residual update module, and a solution update module, directly corresponding to the three blocks in the fine-grid or coarse-grid computation shown in Figure 3.5(c). The iteration module leverages MAC SRAM to perform iterative layer update in 5b to solve for the errors, $\mathbf{e}$ or $\mathbf{e}^*$. After a round of iterations, the residual update module is called to apply the errors in updating the residuals, $\mathbf{r}$ or $\mathbf{r}^*$; and the solution update module is called to accumulate 5b errors to update the full-precision $\mathbf{u}$ values in memory.

Shown in Figure 3.8, the iteration module consists of 4 MAC SRAM arrays. A pair of arrays are used together as the even and odd array to support the rotation mapping. In the prototype design, each array consists of 320×64 8T SRAM cells that are completed with peripherals. The PDE iteration module can be used to compute two independent grids of up to 64×128 (5b grid values), or they can be joined to support a grid of up to 128×128 (5b grid values). The precision is configurable from

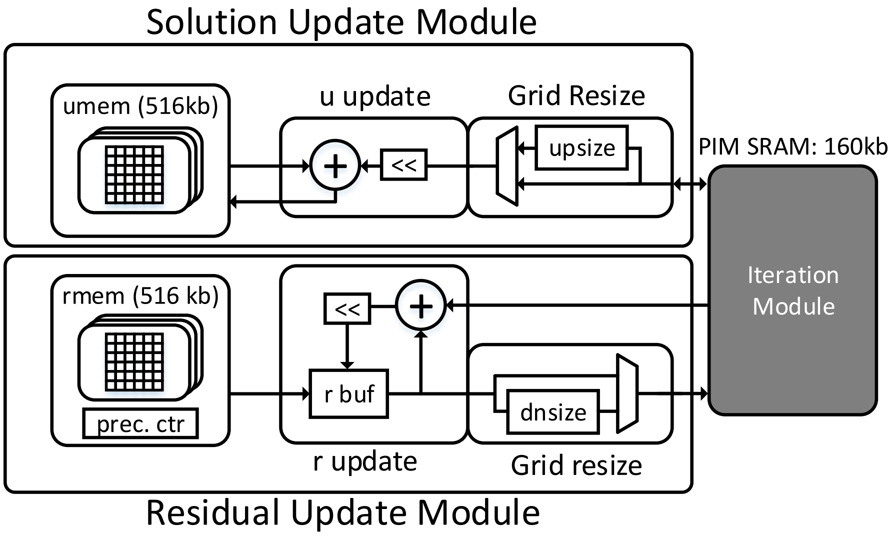


Figure 3.7: Top-level architecture of PDE solver.

1b to 5b. A separate memory is used to store the offsets **c**. Offset subtraction is done at the output of each array. A buffer is added to store the updated solutions for writing back to the neighboring even or odd array.

A 320×64 MAC SRAM array is internally split into two 320×32 banks to be used as the left and right bank, as shown in Figure 3.9. The MAC SRAM array occupies 0.467mm^2 in 180nm CMOS and is clocked at 200MHz. It provides two ports: a single read/write port for normal memory access, and a group read port for MAC operations. In the MAC mode, up to 20 WLs (i.e., four 5b groups, two for each bank) are selected in parallel by the group decoder. 5b width-modulation of WL is controlled by a DLL, and 5b level-modulation is done via current mirrors. Select muxes allow the analog summation of partial sums from the two banks. The 32 merged BLs are digitized by 32 5b ADCs.

Figure 3.10(a) illustrates the timing diagram for the Jacobi update method on a 128×128 grid. The process calculates two rows at a time (top and bottom), and takes 63 rounds of row updates per iteration. Row computation is similar in the hybrid method. Shown in Figure 3.10(b), the hybrid method performs non-blocking updates by writing back for each half-row computation. This grid update process is repeated

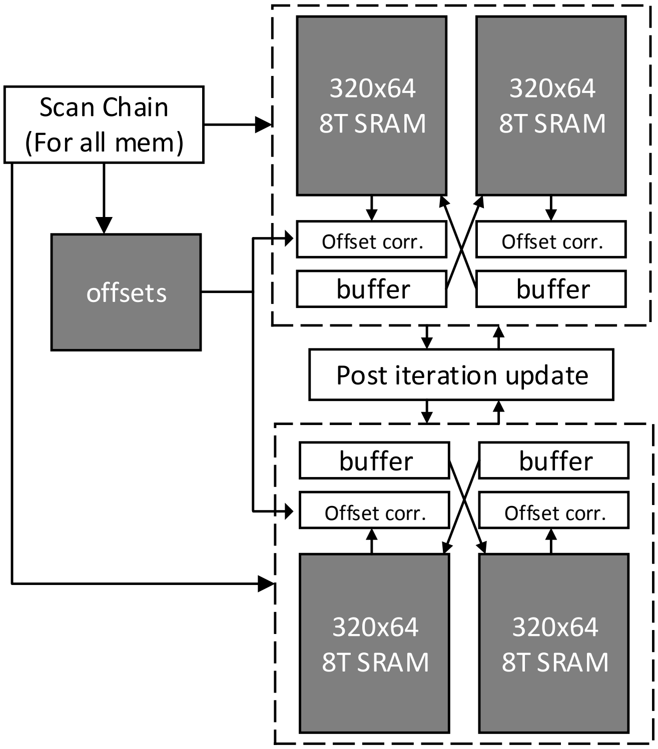


Figure 3.8: Architectural sketch of PDE iteration module.

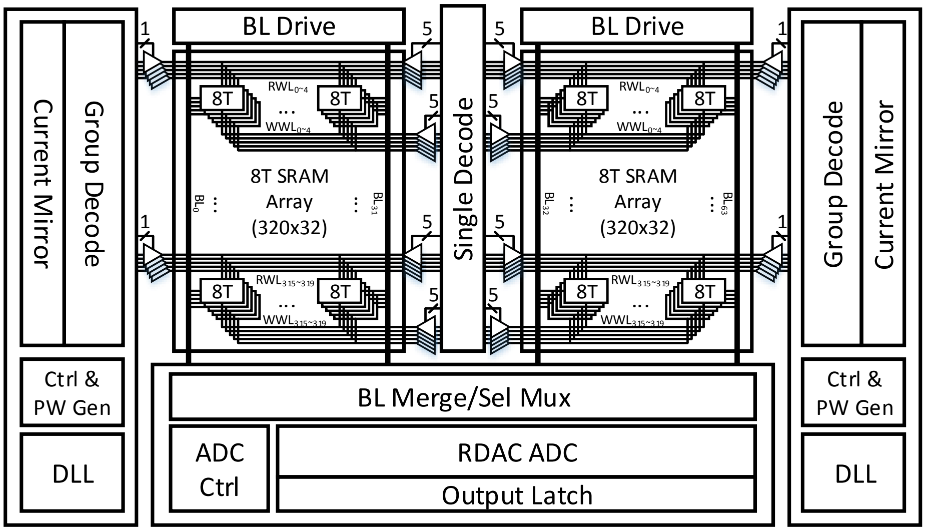


Figure 3.9: Block diagram of MAC SRAM.

until the desired iteration limit is reached.

3.6 Group Read and Word Line Pulse Generation

The group read mode is illustrated in Figure 3.11, showing 4 stencil entries (s_1, s_2, s_3, s_4) applied to 4 error vectors (e_1, e_2, e_3, e_4) stored in 20 rows of the SRAM banks for MAC operations. The MAC operations are conducted in groups in the following manner: 1) the group decoder turns on the access to a group of 5 SRAM rows; 2) the WL pulse width (PW) is selected by a 5b stencil entry; 3) current mirrors generate the WL voltage needed to provide 1×, 2×, 4×, 8×, and 16× unit cell current for the analog readout of 5b error values; and 4) the products between the stencil entry and the error values are accumulated on the BLs. Up to 4 groups are activated at the same time to enable 128 5b×5b MACs in parallel in the MAC SRAM.

If we use the 5ns clock period as the unit pulse width (PW), a 5b WL pulse will take 32 clock cycles, or 160ns. To improve performance, we use 625ps as the unit PW, so a 5b WL pulse only takes 20ns. To generate fine and well-controlled WL pulses, we design a DLL to subdivide a 5ns clock period to a 625ps unit pulse width (PW) using an 8-stage voltage-controlled delay line in a control loop and a pulse generator logic as shown in Figure 3.12. The phases are continuously adjusted by tracking the 200MHz reference clock using the phase detector, and errors are corrected by the control voltage of the delay line.

We allocate up to 200mV BL swing to represent the readout of one group. With all four groups activating at the same time, the BL swings up to 800mV, from 1.8V down to 1.0V. The swing is limited by the one-stage pre-amplifier of the ADC. The pre-amplifier performs offset cancellation at its output for a lower area and power, but the input common-mode range is more limited. Since 800mV BL swing is digitized by a 5b ADC, an LSB step is 25mV.

The process variation is evaluated by Monte Carlo simulations for reading a group

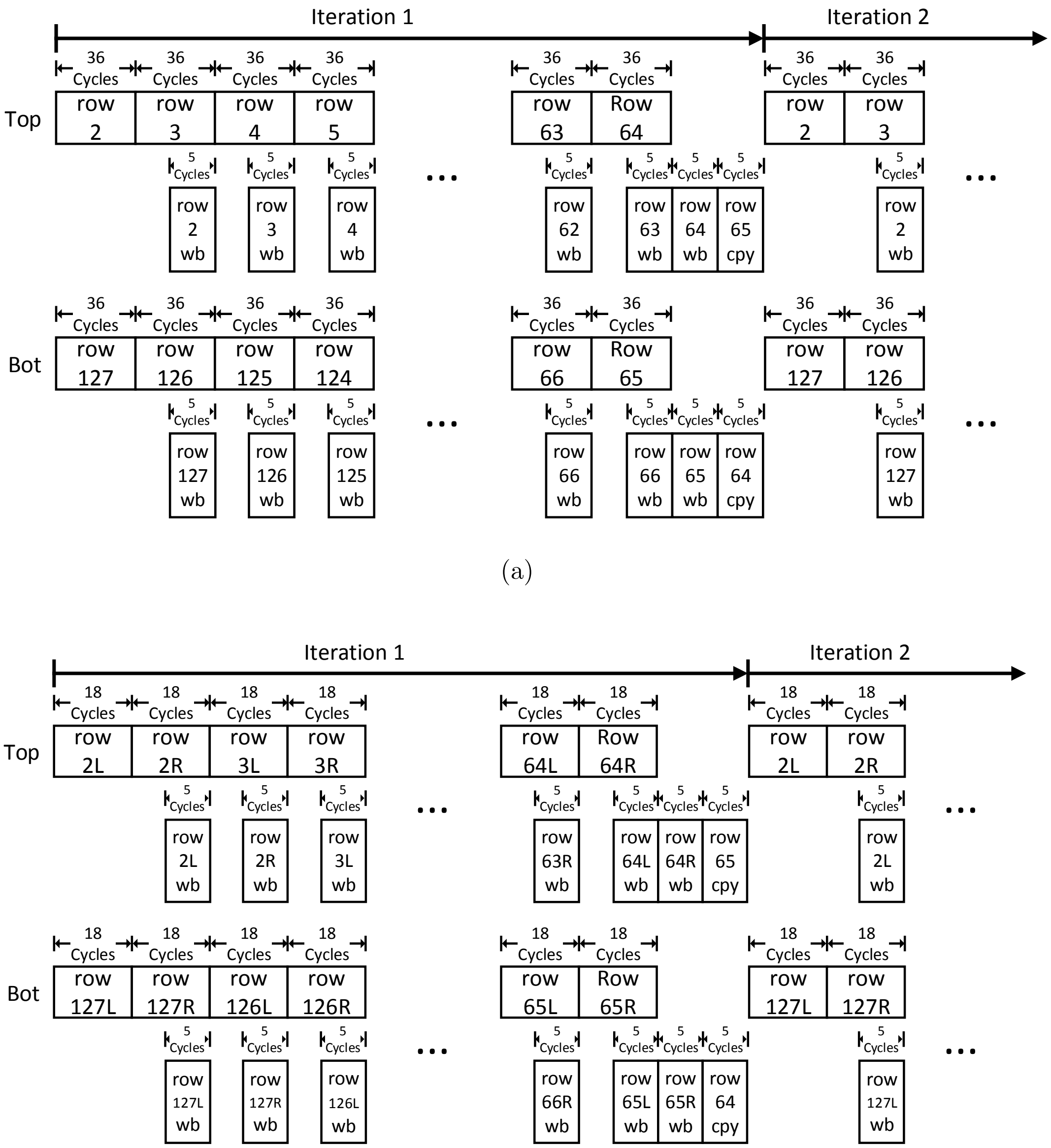


Figure 3.10: Timing diagram of PDE iteration module in (a) the Jacobi method and (b) the hybrid method.

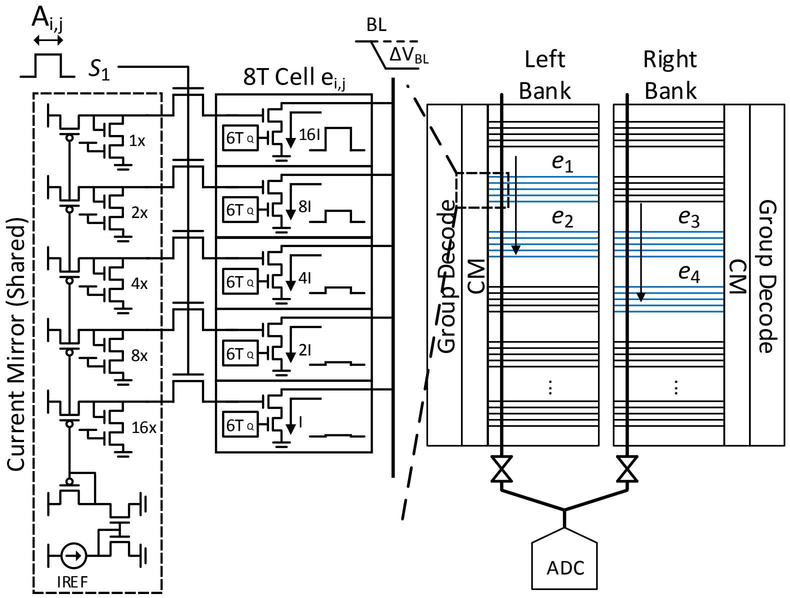


Figure 3.11: Illustration of group read for MAC operations.

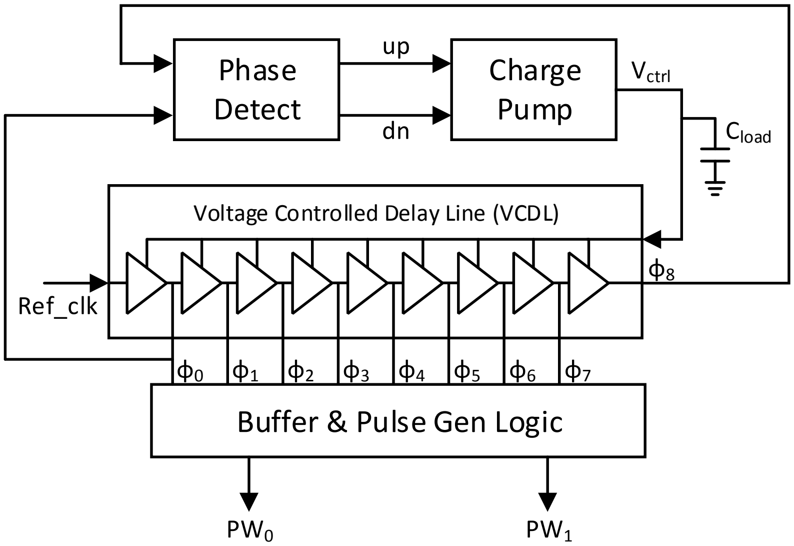


Figure 3.12: DLL design for WL pulse generation.

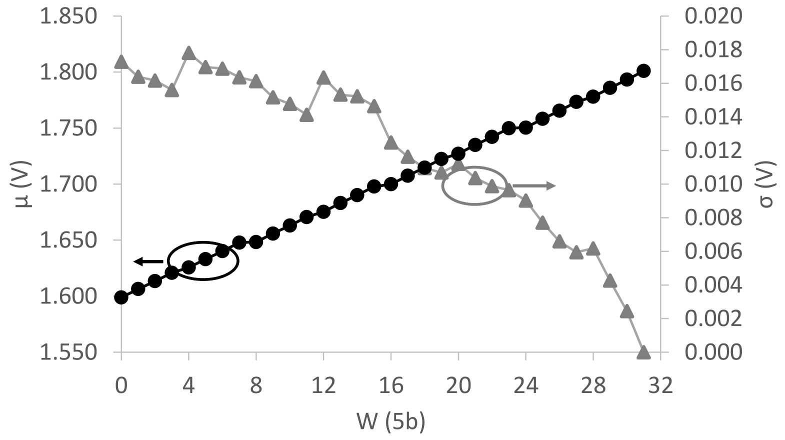


Figure 3.13: BL voltage variation.

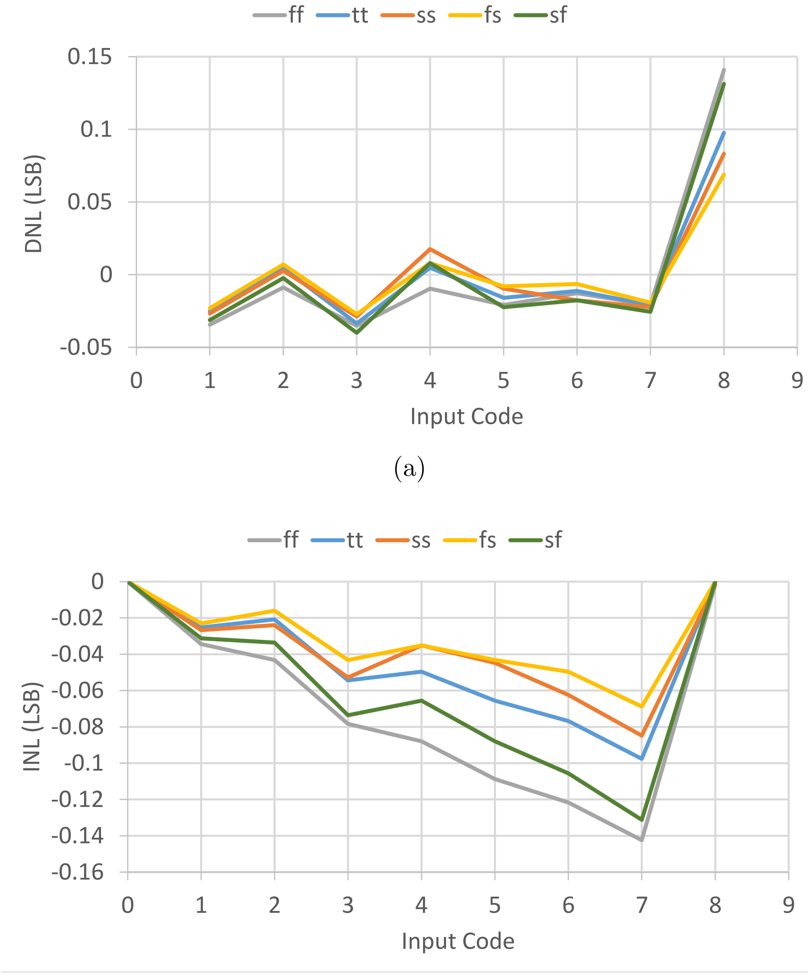
of 5b operands. To obtain the maximum absolute variation, we used the maximum PW of 19.375ns (31 unit PW). The BL voltage and its standard deviation are shown in Figure 3.13. The standard deviation of the BL voltage is shown to be limited to 18mV.

The DLL occupies 1,500μm^2 in 180nm and consumes 950μW. The differential nonlinearity (DNL) and the integral nonlinearity (INL) for the DLL are evaluated for all process corners. The DLL provides a maximum INL of sub-0.15 unit PW, as shown in Figure 3.14. The closed-loop pulse generation is more robust than an open-loop approach [29].

3.7 Bit Line Readout

The BL ADC needs to be compact and energy-efficient to avoid becoming a bottleneck of the design. Therefore, flash or SAR architectures are excluded. Instead, we choose a ramp ADC that consists of a ramp reference and a counter shared by all columns, and a single comparator and latch per column. The ramp architecture minimizes the area and energy, but a 5b conversion requires 32 time steps.

We adopt a compact dual-ramp single-slope (DRSS) ADC architecture [67] that



(a)

(b)

Figure 3.14: Differential nonlinearity (DNL) and integral nonlinearity (INL) of WL pulses generated by DLL.

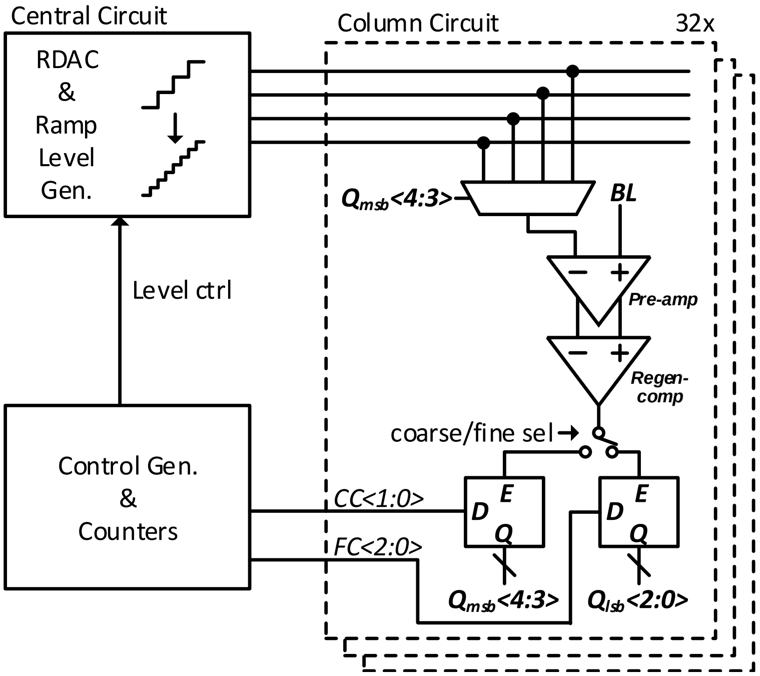
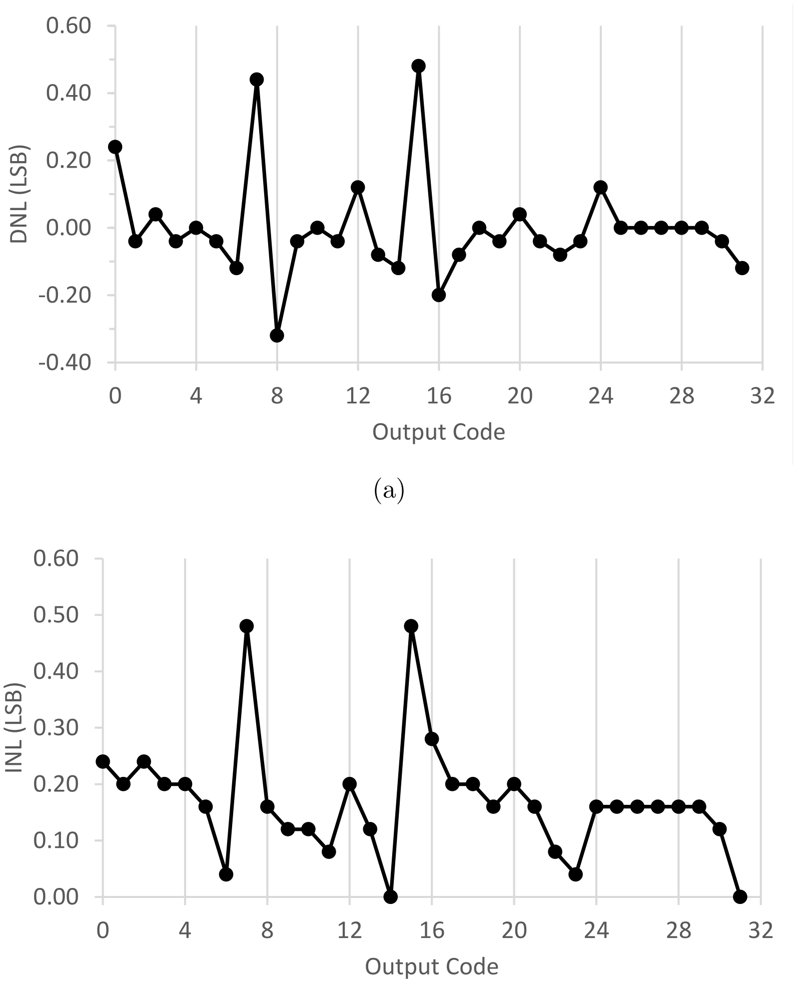


Figure 3.15: 5-bit dual-ramp single-slope ADC design for BL readout.

applies a 2b coarse-ramp comparison followed by a 3b fine-ramp comparison, as shown in Figure 3.15. The BL voltage is first compared with the 2b coarse ramp to obtain the 2b MSB, which then selects one of four 3b fine ramps for comparison to obtain the 3b LSB. The dual-ramp approach performs 5b comparison in $2^2 + 2^3 = 12$ time steps, faster than a serial conversion architecture [29].

In implementing DRSS ADCs, a central circuit is shared by 32 columns and it generates two ramps by a resistive DAC and a controller that steps through the two conversion phases. A compact column circuit consists of a pre-amplifier followed by a regenerative comparator and latches.

The column circuit measures only 350μm × 110μm. The 32 ADCs in a MAC SRAM occupy 0.044mm^2 and the conversion costs 8.91mW at 200MHz. The DNL of the ADC is kept below 0.45b, and the INL of the ADC is within 0.5b as shown in Figure 3.16.



(a)

(b)

Figure 3.16: ADC differential nonlinearity (DNL) and integral nonlinearity (INL).

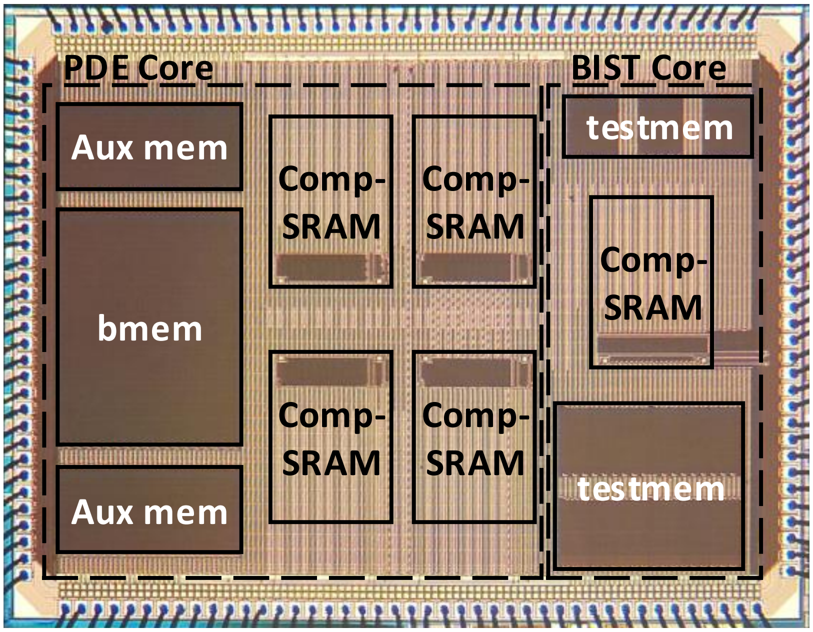


Figure 3.17: Microphotograph of PDE solver test chip.

3.8 Results and Comparison

A 180nm 11.0mm^2 PDE solver test chip was fabricated and tested. The chip consists of a PDE solver and BIST circuits, as shown in Figure 3.17. The 4 MAC SRAMs in the PDE solver core each takes 570μm×820μm and dissipates 16.6mW when performing group read at 200MHz and room temperature.

The ADC, DLL and group decoder account for 62%, 12% and 9% of the power consumption shown in Figure 3.18. When running Jacobi and the hybrid layer update iterations, the 5b multigrid PDE solver reaches an error tolerance of 10^{-8} while speeding up convergence by 6× and 8× respectively, over the baseline 32b single-grid implementation, as shown in Figure 3.19.

The 200MHz MAC SRAM completes 128 5b×5b MAC operations in 18 clock cycles (4-cycle WL pulse, 1-cycle BL propagation, 12-cycle ADC and 1-cycle latching). With 4 MAC SRAMs, the PDE solver chip performs 512 5b×5b MAC operations every 18 clock cycles. Following [29] that counts an operation at each active SRAM cell as 2 OPs, the performance and energy of each MAC SRAM are 14.2GOPS and

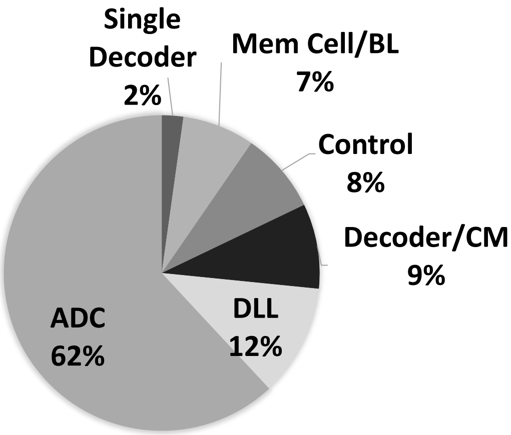


Figure 3.18: Breakdown of MAC SRAM power.

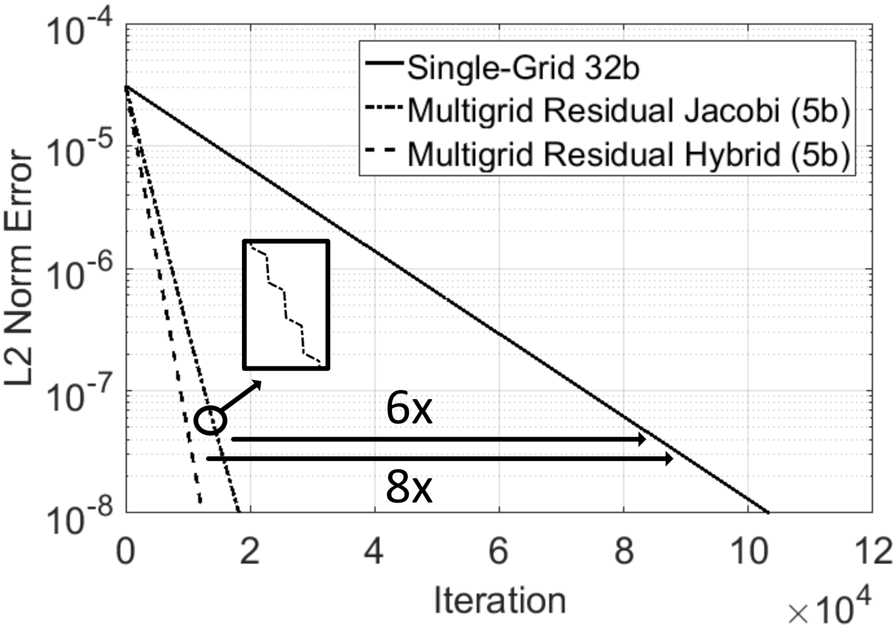


Figure 3.19: Convergence of 5b multigrid (5b fine-grid: 127×127, 5b coarse-grid: 64×64) using Jacobi and hybrid layer update iterations compared to single-grid baseline implementation (32b fine-grid: 127×127). Results are based on solving 2D Poisson's equation.

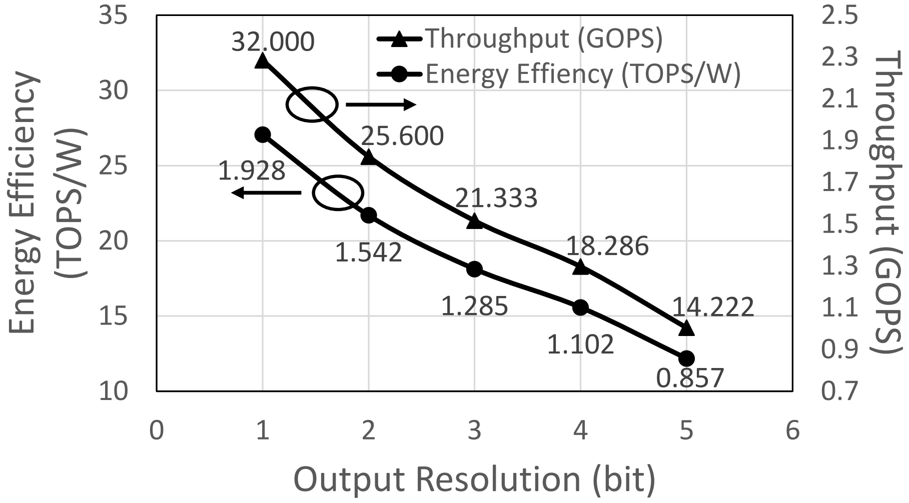


Figure 3.20: Measured performance and energy.

857GOPS/W, respectively. At a lower precision, the performance and energy efficiency can be more than doubled, as shown in Figure 3.20. We synthesized a comparable digital ASIC in 180nm CMOS, but its compute density is 40× lower than this work.

This design is the first PIM that targets solving PDEs. Prior PIM designs do not meet the requirements of the PDE solver due to limited multiplicand precision [27, 29, 30], limited ADC resolution [27, 30], or limited number of ADCs [65, 66]. In Table 3.1, we attempt to compare the PDE solver chip with three SRAM-based PIM designs, the 65nm IMCORE [66], the 65nm Conv-RAM [29] and the 55nm T8T SRAM [72]. Our MAC SRAM provides a higher multiplicand precision than Conv-RAM and substantially more ADCs than IMCORE, Conv-RAM and T8T SRAM to support iterative solution updates. As a result, the power efficiency measured in TOPS/W is lower. Note that the work was prototyped in a 180nm technology and the results in Table 3.1 are not normalized. We expect that the energy efficiency and compute density of this design to improve in newer technologies.

In Table 3.2, this work is compared with recently-published efficient accelerators for solving PDEs: an analog computer accelerator [63], a hybrid computing unit [64],

Table 3.1: Comparison with State-of-the-Art SRAM-Based PIM (Unnormalized)

	IMCORE [66]	Conv-RAM [29]	T8T SRAM [72]	This work
Application	SVM	CNN	CNN	PDE solver
Technology	65nm	65nm	55nm	180nm
Core voltage	0.925V	1.2V	1.0V	1.8V
PIM core area (mm^2)	0.52	0.067	N/A	1.868
Memory size	512×256	16×16×64	64×60	4×320×64
PIM kernel precision	4b×8b	7b×1b	4b×5b	**5b×5b**
ADC resolution	4b	7b	7b	5b
Number of ADCs	1	16	12	**128**
Activated cells	4×256	16×1×64	32×1×60[d]	4×20×32
Latency (ns)	31	150	10.2	90
Performance (GOPS[a])	66.2[b]	10.7[c]	376[d]	56.8
Power (mW)	3.17	0.381	0.960	66.4
Efficiency (TOPS/W)	20.9	28.1	392[d]	0.857
Density (GOPS/mm^2)	127	160	N/A	30.5

[a]An actived cell is counted as 2 OPs [29]. [b]In test mode [66].

[c]Activates 50×16 cells and uses 4b output for testing [29].

[d]Activates 32×60 cells in high precision mode. [29].

Table 3.2: Comparison with Prior PDE Accelerators (Unnormalized)

	Cowan [63]	Guo [64]	Kung [62]	This work
Technology	250nm	65nm	15nm	180nm
Design	Silicon	Silicon	Synthesis	Silicon
Core voltage	2.5V	1.2V	0.8V	1.8V
Core area (mm^2)	100	3.8	0.45	**1.868**
# Active PEs	400[a]	26[a]	64[b]	**512**[b]
Compute precision	8b	16b	32b	5b
Core frequency	20kHz	25kHz	600MHz	200MHz
Peak Grid Update Rate (MEntries/s)	1.6[c]	0.04[c]	66.7	**1420**
Power (mW)	300	1.2	523	66.4
Efficiency (MEntries/s/W)	0.005	0.033	0.127	**21.4**
Density (MEntries/s/mm^2)	0.016	0.01	148	**760**

[a]1 PE represents an analog operation block

[b]1 PE represents a 5b multiplier unit for MAC operation

[c]Assumes outputs are continuously transferred at full bandwidth.

and a digital hardware accelerator [62]. This work is the first to use PIM in PDE applications. It is also the first among hardware accelerators to use a multigrid residual approach to reduce the core precision requirement to 5b. An intrinsic benefit of PIM is less data movement, compared to the other work that relies on frequent accesses of external DRAMs. Optimized towards dense, low-precision compute, this work achieves a grid update rate of 1.42 G entries/s and an energy efficiency of 21.4 M entries/s/W. The computational density and the energy efficiency of this 180nm design are 5.1× and 168× higher than the state-of-the-art 15nm synthesis [62] without technology normalization.

3.9 Conclusion

Numerical PDE solvers require high-precision, iterative and memory-intensive computation. In this work, we adopt a residual form of the multigrid method to reduce the precision requirement, and a row-by-row update to reduce the computation time while providing sufficient parallelism.

The resulting PDE solver design is mapped to a 5b SRAM-based PIM system that consists of an iteration module, a solution update module and a residual update module. Quantized grid values are mapped to SRAM following a rotation mapping method for high storage utilization and efficient parallel computation. Four 320×64 SRAMs perform parallel 5b×5b MAC operations, with 5b word line level modulation and 5b pulse width modulation. Each MAC SRAM output is digitized by 32 5b DRSS ADCs.

The PDE solver is prototyped in 11mm^2 180nm test chip. The chip is measured to achieve a grid update rate of l1.42 G entries/s at 200MHz at a power consumption of 66.4mW. Compared to previously published PDE solver accelerators, this work demonstrates two orders of magnitude improvement in energy efficiency and at least 5× higher compute density without technology normalization. The results show the promise of using PIM in numerical PDE solver applications.

CHAPTER IV

Conclusion

Technology scaling has enabled a wide range of data-intensive applications, and the trend of parallel computing would continue to grow for the foreseeable future. To further improve processing capability that is currently limited by memory bandwidth, one promising approach is to extract and process only the meaningful information, and the other approach is to integrate computation with the memory. This thesis work provides new solutions targeting high-performance and energy-efficient accelerator design for data-intensive applications. Specifically, this work presents two primary approaches to address the von Neumann bottleneck: 1) reducing the amount of data that need to be moved by sparsity and data compression; and 2) practical and robust multibit-memory compute design to extend the applicability of in-memory compute to a wider range of applications. The approaches are demonstrated in two data-intensive applications, a video sequence inference processor and a PDE solver.

A video sequence inference processor computes on video inputs and 3D features that are inherently more complex than image processing. We applied multiple approaches to enhance the data sparsity and reduce the computational complexity. Using the residual form, we demonstrate the algorithm reformulation that leads to an all-spiking three-layer architecture, all implemented primarily in SAs, instead of MACs. The resulting data sparsity reaches a high 90% level. By applying kernel compression and activation compression, memory size can be reduced further by 43% and 64%, respectively. Applying both techniques increases the processing throughput and reduces the energy by 51× and 63×, respectively, while the area is kept nearly constant. These optimization are essential for enabling the processing of 1080p HD videos at 60 fps. The design was demonstrated in a 2.53mm^2 40nm CMOS chip, achieving 1.70TOPS at 0.9V and 250MHz at a power dissipation of 135mW. Tested with the 6-class KTH Human Action Dataset, the chip provides a 76.7% classification accuracy.

This work shows that the naturally occurring data sparsity could be leveraged to design an architecture with reduced data size and simplified computations. These lead to reduced memory bandwidth to address the von Neumann bottleneck, and are essential for applications that require processing of large sparse data sets.

A PDE solver is an important scientific computation problem that is challenged by big data and its high precision requirement. PDE solvers are most frequently implemented on GPUs and even supercomputers. To reduce the problem size, we adopted a residual form to quantize the the floating-point compute problem to 5b fixed-point compute, by computing based on the incremental differences. In conjunction, we applied the multigrid method and a row-by-row update method to speed up convergence. These optimization methods lead to a fast PDE solver design that operates primarily in low precision, which could be mapped to in-memory computing. The quantized grid values are rotationally mapped to the MAC SRAM for high storage utilization and efficient parallel computation. The MAC SRAM is designed to compute 5b×5b MAC with high parallelism. To support multi-bit computation, we adopt both width and level modulation of word-line pulses. To reduce the cost and improve the speed of analog-to-digital conversion, we employ a compact array of dual-ramp single-slope (DRSS) ADCs for bit-line readout.

These approaches lead to a PDE solver prototyped in a 11mm^2 180nm CMOS chip. The chip integrates four 320×64 MAC SRAMs, each capable performing 128× parallel 5b×5b MACs with 32 5-bit output ADCs and consuming 16.6mW at 200MHz. The design provides 40× compute density when compared to an equivalent ASIC. The prototype chip is measured to solve a 127×127 PDE grid with a grid update rate of 1.42 G entries/s at 200MHz and power consumption of 66.4mW. When compared to other PDE accelerators, this work achieves two orders of magnitude better energy efficiency and more than 5× higher compute density.

This work demonstrates the use of algorithm reformulation, precision reduction, and in-memory computing to design a highly efficient PDE solver, where the von Neumann bottleneck is removed entirely.

To extend PDE solver to wider applications, some PDE models require non-uniform or even unstructured discretization to speed up convergence and reach a higher accuracy. However, such approaches would result in non-uniform stencil matrices, posing challenges in computation parallelism and storage density. New memory designs that provide more flexibility in bit cell access patterns would be required to support such approaches.

Applying a multigrid method on non-uniform or unstructured grids requires an alternative approach that does not rely on geometric information. Thus, algebraic multigrid (AMG) methods have been introduced, where grid coarsening is based on the matrix system itself [73, 74]. Because AMG aims to generalize grid structures for coarsening, multiple classes of AMG methods have been introduced to optimize the formulation of the approximate compute. One notable approach is aggregation based AMG [75], which decomposes stencil matrix in the coarsening process. Another approach is adaptive AMG [76], which optimizes parameters based on iteration results. These approaches in approximate computing provide new opportunities to design highly parallel and compact PDE solvers.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] L. Null and J. Lobur, *Essentials of Computer Organization and Architecture*. Jones & Bartlett Learning, 2018. [Online]. Available: https://books.google.com/books?id=GldUDwAAQBAJ

[2] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *CoRR*, vol. abs/1605.07678, 2016. [Online]. Available: http://arxiv.org/abs/1605.07678

[3] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on.* IEEE, 2017, pp. 1–12.

[4] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on.* IEEE, 2016, pp. 380–392.

[5] M. Kang, S. K. Gonugondla, A. Patil, and N. R. Shanbhag, "A multi-functional in-memory inference processor using a standard 6t sram array," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 2, pp. 642–655, 2018.

[6] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International.* IEEE, 2014, pp. 10–14.

[7] J. Von Neumann, "First draft of a report on the edvac," *IEEE Annals of the History of Computing*, no. 4, pp. 27–75, 1993.

[8] D. C. Brock and G. E. Moore, *Understanding Moore's law: four decades of innovation.* Chemical Heritage Foundation, 2006.

[9] C. Martin, "Multicore processors: challenges, opportunities, emerging trends," in *Proc. Embedded World Conference*, vol. 2014, 2014, p. 1.

[10] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[11] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam *et al.*, “Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, 2015.

[12] K. Ehsani, H. Bagherinezhad, J. Redmon, R. Mottaghi, and A. Farhadi, “Who let the dogs out? modeling dog behavior from visual data,” *arXiv preprint arXiv:1803.10827*, 2018.

[13] M. Kanakaraj and R. M. R. Guddeti, “Performance analysis of ensemble methods on twitter sentiment analysis using nlp techniques,” in *Semantic Computing (ICSC), 2015 IEEE International Conference on.* IEEE, 2015, pp. 169–170.

[14] Y. LeCun, E. Cosatto, J. Ben, U. Muller, and B. Flepp, “Dave: Autonomous off-road vehicle control using end-to-end learning,” Technical Report DARPA-IPTO Final Report, Courant Institute/CBLL, http://www. cs. nyu. edu/yann/research/dave/index. html, Tech. Rep., 2004.

[15] Z. Lähner, D. Cremers, and T. Tung, “Deepwrinkles: Accurate and realistic clothing modeling,” in *European Conference on Computer Vision.* Springer, 2018, pp. 698–715.

[16] A. Clark and A. Dünser, “An interactive augmented reality coloring book,” in *2012 IEEE Symposium on 3D User Interfaces (3DUI).* IEEE, 2012, pp. 7–10.

[17] M. B. Taylor, “The evolution of bitcoin hardware,” *Computer*, no. 9, pp. 58–66, 2017.

[18] L. Kan, Y. Wei, A. H. Muhammad, W. Siyuan, G. Linchao, and H. Kai, “A multiple blockchains architecture on inter-blockchain communication,” in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C).* IEEE, 2018, pp. 139–145.

[19] Nvidia, “Nvidia tesla v100 tensor core gpu,” https://www.nvidia.com, 2018.

[20] J. Backus, *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs.* ACM, 2007.

[21] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, “A domain-specific architecture for deep neural networks,” *Communications of the ACM*, vol. 61, no. 9, pp. 50–59, 2018.

[22] D. Patterson, “50 years of computer architecture: From the mainframe cpu to the domain-specific tpu and the open risc-v instruction set,” in *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International.* IEEE, 2018, pp. 27–31.

[23] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra, "A review of near-memory computing architectures: Opportunities and challenges," in *Proceedings of the 21st Euromicro Conference on Digital System Design (DSD)*, 2018.

[24] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin *et al.*, "25.2 a 1.2 v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International.* IEEE, 2014, pp. 432–433.

[25] D.-I. Jeon, K.-B. Park, and K.-S. Chung, "Hmc-mac: Processing-in memory architecture for multiply-accumulate operations with hybrid memory cube," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 5–8, 2018.

[26] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 751–764, 2017.

[27] J. Zhang, Z. Wang, and N. Verma, "In-memory computation of a machine-learning classifier in a standard 6T SRAM array," *IEEE J. Solid-State Circuits*, vol. 52, no. 4, pp. 915–924, 2017.

[28] C.-C. Chou, Z.-J. Lin, P.-L. Tseng, C.-F. Li, C.-Y. Chang, W.-C. Chen, Y.-D. Chih, and T.-Y. J. Chang, "An n40 256k× 44 embedded rram macro with sl-precharge sa and low-voltage current limiter to improve read and write performance," in *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International.* IEEE, 2018, pp. 478–480.

[29] A. Biswas and A. P. Chandrakasan, "Conv-RAM: An energy-efficient SRAM with embedded convolution computation for low-power CNN-based machine learning applications," in *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2018, pp. 488–490.

[30] W.-S. Khwa, J.-J. Chen, J.-F. Li, X. Si, E.-Y. Yang, X. Sun, R. Liu, P.-Y. Chen, Q. Li, S. Yu, and M.-F. Chang, "A 65nm 4Kb algorithm-dependent computing-in-memory SRAM unit-macro with 2.3 ns and 55.8 TOPS/W fully parallel product-sum operation for binary dnn edge processors," in *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2018, pp. 496–498.

[31] D. L. Donoho, "Compressed sensing," *IEEE Transactions on information theory*, vol. 52, no. 4, pp. 1289–1306, 2006.

[32] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete cosine transform," *IEEE transactions on Computers*, vol. 100, no. 1, pp. 90–93, 1974.

[33] B. A. Olshausen and D. J. Field, "Emergence of simple-cell receptive field properties by learning a sparse code for natural images," *Nature*, vol. 381, no. 6583, p. 607, 1996.

[34] S. Shapero, C. Rozell, and P. Hasler, "Configurable hardware integrate and fire neurons for sparse approximation," *Neural Networks*, vol. 45, pp. 134–143, 2013.

[35] B. Olshausen, "Learning sparse, overcomplete representations of time-varying natural images," in *International Conference on Image Processing*, vol. 1. IEEE, 2003, pp. I–41–44.

[36] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in neural information processing systems*, 2016, pp. 4107–4115.

[37] R. E. Schapire and Y. Freund, *Boosting: Foundations and algorithms.* MIT press, 2012.

[38] Y. Saad, *Iterative methods for sparse linear systems.* siam, 2003, vol. 82.

[39] M. Kang, S. K. Gonugondla, and N. R. Shanbhag, "A 19.4 nJ/decision 364K decisions/s in-memory random forest classifier in 6T SRAM array," in *European Solid-State Circuits Conf. (ESSCIRC)*, 2017, pp. 263–266.

[40] J. Oh, G. Kim, J. Park, I. Hong, S. Lee, J.-Y. Kim, J.-H. Woo, and H.-J. Yoo, "A 320 mw 342 gops real-time dynamic object recognition processor for hd 720p video streams," *IEEE J. Solid-State Circuits*, vol. 48, no. 1, pp. 33–45, 2013.

[41] G. Kim, J. Oh, S. Lee, and H.-J. Yoo, "An 86 mw 98gops ann-searching processor for full-hd 30 fps video object recognition with zeroless locality-sensitive hashing," *IEEE J. Solid-State Circuits*, vol. 48, no. 7, pp. 1615–1624, 2013.

[42] K. J. Lee, G. Kim, J. Park, and H.-J. Yoo, "A vocabulary forest object matching processor with 2.07 m-vector/s throughput and 13.3 nj/vector per-vector energy for full-hd 60 fps video object recognition," *IEEE J. Solid-State Circuits*, vol. 50, no. 4, pp. 1059–1069, 2015.

[43] D. Jeon, M. B. Henry, Y. Kim, I. Lee, Z. Zhang, D. Blaauw, and D. Sylvester, "An energy efficient full-frame feature extraction accelerator with shift-latch fifo in 28 nm cmos," *IEEE J. Solid-State Circuits*, vol. 49, no. 5, pp. 1271–1284, 2014.

[44] A. Suleiman, Z. Zhang, and V. Sze, "A 58.6 mw 30 frames/s real-time programmable multiobject detection accelerator with deformable parts models on full hd 1920 × 1080 videos," *IEEE J. Solid-State Circuits*, vol. 52, no. 3, pp. 844–855, 2017.

[45] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proc. IEEE Int. Conf. on Computer Vision*, vol. 2, 1999, pp. 1150–1157.

[46] H. Bay, T. Tuytelaars, and L. Van Gool, “Surf: Speeded up robust features,” in *Proc. European Conf. Computer Vision*, 2006, pp. 404–417.

[47] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, “Object detection with discriminatively trained part-based models,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 9, pp. 1627–1645, 2010.

[48] P. Dollár, V. Rabaud, G. Cottrell, and S. Belongie, “Behavior recognition via sparse spatio-temporal features,” in *Int. Workshop on Visual Surveillance and Performance Evaluation of Tracking and Surveillance*, 2005, pp. 65–72.

[49] I. Laptev, M. Marszalek, C. Schmid, and B. Rozenfeld, “Learning realistic human actions from movies,” in *Proc. IEEE Int. Conf. Computer Vision and Pattern Recognition*, 2008, pp. 1–8.

[50] G. Willems, T. Tuytelaars, and L. Van Gool, “An efficient dense and scale-invariant spatio-temporal interest point detector,” in *Proc. European Conf. Computer Vision*, 2008, pp. 650–663.

[51] M. Baccouche, F. Mamalet, C. Wolf, C. Garcia, and A. Baskurt, “Spatio-temporal convolutional sparse auto-encoder for sequence classification.” in *BMVC*, 2012, pp. 1–12.

[52] C. J. Rozell, D. H. Johnson, R. G. Baraniuk, and B. A. Olshausen, “Sparse coding via thresholding and local competition in neural circuits,” *Neural computation*, vol. 20, no. 10, pp. 2526–2563, 2008.

[53] P. Knag, J. K. Kim, T. Chen, and Z. Zhang, “A sparse coding neural network asic with on-chip learning for feature extraction and encoding,” *IEEE J. Solid-State Circuits*, vol. 50, no. 4, pp. 1070–1079, 2015.

[54] C. Liu, S.-G. Cho, and Z. Zhang, “A 2.56-mm^2 718gops configurable spiking convolutional sparse coding accelerator in 40-nm cmos,” *IEEE J. Solid-State Circuits*, vol. 53, no. 10, pp. 2818–2827, 2018.

[55] S. Savarese, A. DelPozo, J. C. Niebles, and L. Fei-Fei, “Spatial-temporal correlatons for unsupervised action classification,” in *Proc. IEEE Workshop Motion and Video Computing*, 2008, pp. 1–8.

[56] K. Zhang, L. Zhang, and M.-H. Yang, “Real-time compressive tracking,” in *Proc. European Conf. Computer Vision*, 2012, pp. 864–877.

[57] P. F. Schultz, D. M. Paiton, W. Lu, and G. T. Kenyon, “Replicating kernels with a short stride allows sparse reconstructions with fewer independent kernels,” *arXiv preprint arXiv:1406.4205*, 2014.

[58] C. Schuldt, I. Laptev, and B. Caputo, “Recognizing human actions: a local svm approach,” in *Proc. IEEE Int. Conf. Pattern Recognition*, vol. 3, 2004, pp. 32–36.

[59] V. Veeriah, N. Zhuang, and G.-J. Qi, “Differential recurrent neural networks for action recognition,” in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 4041–4049.

[60] R. S. Varga, *Matrix Iterative analysis, 2nd edition.* Springer, 2000, vol. 27.

[61] Y. Huang, N. Guo, M. Seok, Y. Tsividis, and S. Sethumadhavan, “Evaluation of an analog accelerator for linear algebra,” in *Int. Symp. on Computer Architecture (ISCA)*, vol. 44, no. 3, 2016, pp. 570–582.

[62] J. Kung, Y. Long, D. Kim, and S. Mukhopadhyay, “A programmable hardware accelerator for simulating dynamical systems,” in *Int. Symp. on Computer Architecture (ISCA)*, 2017, pp. 403–415.

[63] G. E. Cowan, R. C. Melville, and Y. P. Tsividis, “A vlsi analog computer/digital computer accelerator,” *IEEE J. Solid-State Circuits*, vol. 41, no. 1, pp. 42–53, 2006.

[64] N. Guo, Y. Huang, T. Mai, S. Patil, C. Cao, M. Seok, S. Sethumadhavan, and Y. Tsividis, “Energy-efficient hybrid analog/digital approximate computation in continuous time,” *IEEE J. Solid-State Circuits*, vol. 51, no. 7, pp. 1514–1524, 2016.

[65] M. Kang, S. Gonugondla, A. Patil, and N. Shanbhag, “A 481pJ/decision 3.4 M decision/s multifunctional deep in-memory inference processor using standard 6T SRAM array,” *arXiv preprint arXiv:1610.07501*, 2016.

[66] S. K. Gonugondla, M. Kang, and N. Shanbhag, “A 42pJ/decision 3.12 TOPS/W robust in-memory machine learning classifier with on-chip training,” in *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2018, pp. 490–492.

[67] M. F. Snoeij, P. Donegan, A. J. Theuwissen, K. A. Makinwa, and J. H. Huijsing, “A CMOS image sensor with a column-level multiple-ramp single-slope ADC,” in *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2007, pp. 506–618.

[68] P. Pérez, M. Gangnet, and A. Blake, “Poisson image editing,” *ACM Transactions on graphics (TOG)*, vol. 22, no. 3, pp. 313–318, 2003.

[69] J. Peiró and S. Sherwin, “Finite difference, finite element and finite volume methods for partial differential equations,” in *Handbook of materials modeling.* Springer, 2005, pp. 2415–2446.

[70] Y. Xu, “Hybrid jacobian and gauss–seidel proximal block coordinate update methods for linearly constrained convex programming,” *SIAM Journal on Optimization*, vol. 28, no. 1, pp. 646–670, 2018.

[71] D. M. Young, *Iterative solution of large linear systems.* Elsevier, 2014.

[72] X. Si, J.-J. Chen, Y.-N. Tu, W.-H. Huang, J.-H. Wang, Y.-C. Chiu, W.-C. Wei, S.-Y. Wu, X. Sun, R. Liu, S. Yu, R.-S. Liu, C.-C. Hsieh, K.-T. Tang, Q. Li, and M.-F. Chang, "24.5 a twin-8t sram computation-in-memory macro for multiple-bit cnn-based machine learning," in *IEEE Int. Solid-State Circuits Conf. (ISSCC)*. IEEE, 2019, pp. 396–398.

[73] J. Xu and L. Zikatanov, "Algebraic multigrid methods," *Acta Numerica*, vol. 26, pp. 591–721, 2017.

[74] R. D. Falgout, "An introduction to algebraic multigrid computing," *Computing in science & engineering*, vol. 8, no. 6, p. 24, 2006.

[75] Y. Notay, "An aggregation-based algebraic multigrid method," *Electronic transactions on numerical analysis*, vol. 37, no. 6, pp. 123–146, 2010.

[76] M. Brezina, R. Falgout, S. MacLachlan, T. Manteuffel, S. McCormick, and J. Ruge, "Adaptive smoothed aggregation (α sa) multigrid," *SIAM review*, vol. 47, no. 2, pp. 317–346, 2005.