University of California

Los Angeles

An Energy-Efficient Sparse-BLAS Coprocessor using STT-MRAM

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Electrical Engineering

by

Richard William Dorrance

2015

Abstract of the Dissertation

An Energy-Efficient Sparse-BLAS Coprocessor using STT-MRAM

by

Richard William Dorrance

Doctor of Philosophy in Electrical Engineering

University of California, Los Angeles, 2015

Professor Dejan Marković, Chair

Sparse linear algebra arises in a wide variety of computational disciplines, including medical imaging, 3D graphics, compressive sensing, neural networks, bioinformatics, and various optimization problems. In recent years, tuned software libraries for multi-core microprocessors (CPUs) and graphics processing units (GPUs) have become the status quo for performing sparse linear algebra in high-performance computing (HPC) environments. However, the computational throughput of these libraries for sparse matrices tends to be significantly lower than that of dense matrices, mostly due to the fact that the compression formats required to efficiently store sparse matrices mismatches traditional computing architectures. This presents a problem, particularly in a mobile environment, where consumer demand for smart phones and tablets has dictated ever increasing computational performance on a limited energy budget.

To address this issue, we have carefully modeled the computational efficiency of sparse algorithms on CPUs and GPUs to identify the computational and memory bottlenecks in their architectures. Using this we have developed a sparse linear algebra kernel that is scalable to efficiently utilize the available memory bandwidth and computing resources. Benchmarking results on a Virtex-5 SX95T field-programmable gate array (FPGA) prototype demonstrate an average computational efficiency of 91.85%. The kernel achieves a peak computational efficiency of 99.8%, a >50x improvement over state-of-the-art CPUs and a >300x improvement over state-of-the-art GPUs. In addition, the sparse linear algebra FPGA kernel is able to achieve higher performance than its CPU and GPU counterparts, while using only 64 single-precision processing elements, with an overall 23-30x improvement in energy efficiency.

An ASIC implementation, in a 40nm 1P10M CMOS process, of the sparse linear algebra kernel is able to achieve a maximum performance of 4.12 GFLOP/s. The minimum energy point (190.31 GFLOP/s/W at 0.6V and 160MHz) shows an energy efficiency improvement of more than a 3,073x, 2,262x, and 66.6x over the CPU, GPU, and FPGA implementations, respectively. Additionally, a data stream reordering scheme was able to eliminate over 99% of data hazards in 14 test matrices for an average boost of 20% in computational efficiency over the FPGA implementation. Further improvements in the energy efficiency could be made by replacing the on-chip SRAM with spintronic memories. Fabrication results from three STT-MRAM chips and two MeRAM chips are also reported.

The dissertation of Richard William Dorrance is approved.

University of California, Los Angeles

2015

"I'll be honest – we're throwing science at the wall
here to see what sticks. No idea what it'll do."
—Cave Johnson, *Portal 2*

Table of Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

First, I would like to thank my advisor, Professor Dejan Marković, without whom this dissertation could never have been written. I am sincerely grateful for the help and support he has given me over that past 6 years at UCLA.

This work was only possible due to the infrastructure and support provided by current and former members of the Parallel Data Architectures research group at UCLA. In no particular order, I would like to whole heartedly thank Henry Chen, Yuta Toriyama, Sina Basir-Kazeruni, Hariprasad Chandrakumar, Dejan Rozgic, Vahagn Hokhikyan, Dr. Cheng C. Wang, Dr. Fang-Li Yuan, Dr. Vaibhav Karkare, Dr. Sarah Gibson, Dr. Tsung-Han Yu, Dr. Rashmi Nanda, Dr. Victoria Wang, and Professors Fengbo Ren and Chia-Hsiang Yang. Furthermore, I would also like to thank Dr. Yousr Ismail, Dr. Rodney Chandler, Dr. Mansour Rachid, Neha Sinha, Sameed Hameed, Qaiser Nehal, Alireza Yousefi, Dr. Henry Park, Dr. Amir Amin Hafez, and Preeti Mulage.

I would like to thank colleagues Dr. Juan G. Alzate, Dr. Pramey Upadhyaya, Dr. Sergiy Cherepov, Hochul Lee, Mark Lewis, and Professors Kang L. Wang, Pedram Khalili, and Yaroslav Tserkovnyak for their work in spintronics. They provided an extensive number of discussions, device characterizations, simulations, and support for the development of my MTJ macro-model and the experimental demonstration of my MeRAM crossbar array.

I am eternally grateful to all of the staff in the Electrical Engineering Department for their help behind the scenes, particularly: Kyle Jung, Deeona Columbia, and Mandy Smith.

Most of all, I would like to thank my parents, Gary Dorrance and Karen Lawrence, for all the love and support they have provided over the years. Last, but certainly not least, I would like to thank my brother, William Dorrance, and his adorable cat, Ellie. He's always managed to keep me well grounded and focused despite all of life's obstacles.

And finally, I would like to thank my ER Surgeon, Dr. Daniel Tseng, and the all of the doctors and nurses of the Good Samaritan Medical Center for saving my life. No Ph.D. goes

off without a hitch. In my case, life decided that I'd been having things a little too easy and felt that acute appendicitis (with peritonitis) three weeks before my defense would be an excellent way to build character.

Vita

2001 – 2005	Valhalla High School, El Cajon, California.
2008	Teaching Assistant, EE140: Linear Integrated Circuits, University of California, Berkeley.
2008	ONR NRIEP Intern, SPAWAR Systems Center, San Diego, CA. Signal Analysis, NOAA costal hydrophones arrays.
2009	B.S., Electrical Engineering and Computer Sciences, University of California, Berkeley.
2009 – 2015	Graduate Student Researcher, Department of Electrical Engineering, University of California, Los Angeles.
2011	M.S., Electrical Engineering, University of California, Los Angeles.
2012 – 2014	Teaching Assistant, EEM216A: Design of VLSI Circuits and Systems, University of California, Los Angeles.
2012	University Fellowship, University of California, Los Angeles.
2013	Teaching Assistant, EE215B: Advanced Digital Integrated Circuits, University of California, Los Angeles.
2013	IEEE Solid-State Circuits Society Student Travel Grant
2013	Visiting Scholar, imec, Leuven Belgium. Circuit Designer, STT-RAM development.
2013	SRC Intern, GLOBALFOUNDRIES Inc., Sunnyvale, CA. Circuit Designer, STT-RAM development.
2013	2014 Qualcomm Innovation Fellowship Finalist
2014	2013-2014 Henry Samueli Excellence in Teaching Award

Publications

H. Lee, J.G. Alzate, **R. Dorrance**, D. Marković, P.K. Amiri , and K.L. Wang, "Design of a Fast and Low-Power Sense Amplifier and Writing Circuit for High-Speed MRAM," *IEEE Trans. Magn. (TMAG)*, vol. 51, no. 5, pp. 1-7, May 2015.

P. K. Amiri, **R. Dorrance**, D. Marković, K. L. Wang, "Nonvolatile Magneto-Electric Random Access Memory Circuit with Burst Writing and Back-to-Back Reads," *US Patent*, US 20140071732 A1, Mar. 2014.

P. K. Amiri, **R. Dorrance**, D. Marković, K. L. Wang, "Read-Disturbance-Free Nonvolatile Content Addressable Memory (CAM)," *US Patent*, US 20140071728 A1, Mar. 2014.

R. Dorrance, F. Ren, and D. Marković, "An Efficient Sparse Matrix-Vector Multiplication (SpMxV) Kernel for Sparse-BLAS on FPGAs," in *Proc. 2014 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA'14)*, pp. 161-170, Feb. 2014.

F. Ren, **R. Dorrance**, W. Xu, and D. Marković, "A Single-Precision Compressive Sensing Signal Reconstruction Engine on Reconfigurable Platform," in *Proc. 23rd Int. Conf. on Field-Programmable Logic and Applications (FPL'13)*, pp. 1-4, Sep. 2013.

R. Dorrance, J.G. Alzate, S. Cherepov, P. Upadhyaya, I.N. Krivorotov, J.A. Katine, J. Langer, K.L. Wang, P.K. Amiri, and D. Marković, "A Diode-MTJ Crossbar Memory Cell Using Voltage-Induced Unipolar Switching for High-Density MRAM," *IEEE Electron Device Lett. (EDL)*, vol. 34, no. 6, pp. 753-755, Jun. 2013.

R. Dorrance, J.G. Alzate, S. Cherepov, P. Upadhyaya, K.L. Wang, P.K. Amiri, and D. Marković, "Voltage-Controlled MRAM for 3D Stackable Non-Volatile Memories," *IEEE Int. Solid-State Circuits Conference Student Research Preview (ISSCC'13)*, Feb. 2013.

J. G. Alzate, P.K. Amiri, P. Upadhyaya, S.S. Cherepov, J. Zhu, M. Lewis, **R. Dorrance**, J. A. Katine, J. Langer, K. Galatsis, D. Marković, I. Krivorotov, and K. L. Wang, "Voltage-Induced Switching of Nanoscale Magnetic Tunnel Junctions," in *Proc. Int. Electron Devices Meeting (IEDM'12)*, pp. 29.5.1-29.5.4, Dec. 2012.

R. Dorrance, F. Ren, Y. Toriyama, A.A. Hafez, C.-K.K. Yang, D. Marković, "Scalability and Design-Space Analysis of a 1T-1MTJ Memory Cell for STT-RAM," *IEEE Trans. Electron Devices (TED)*, vol. 59, no. 4, pp. 878-887, Apr. 2012.

F. Ren, H. Park, **R. Dorrance**, Y. Toriyama, A. Amin, C.-K.K. Yang, D. Marković, "A Body-Voltage-Sensing-Based Short Pulse Reading Circuit for Advanced Spin-Torque

Transfer RAMs (STT-RAMs)," in *Proc. 13th Int. Symp. on Quality Electronic Design (ISQED'12)*, pp, 275 282, Mar. 2012.

H. Park, **R. Dorrance**, A. Amin, F. Ren, D. Marković, C.-K.K. Yang, "Analysis of STT-RAM Cell Design with Multiple MTJs Per Access," in *Proc. ACM/IEEE Int. Symp. on Nanoscale Arch. (NANOARCH'11)*, pp. 32-36, Jun. 2011.

R. Dorrance, F. Ren, Y. Toriyama, A. Amin, C.-K.K. Yang, D. Marković, "Scalability and Design-Space Analysis of a 1T-1MTJ Memory Cell," in *Proc. ACM/IEEE Int. Symp. on Nanoscale Arch. (NANOARCH'11)*, pp. 53-58, Jun. 2011.

CHAPTER 1

Introduction

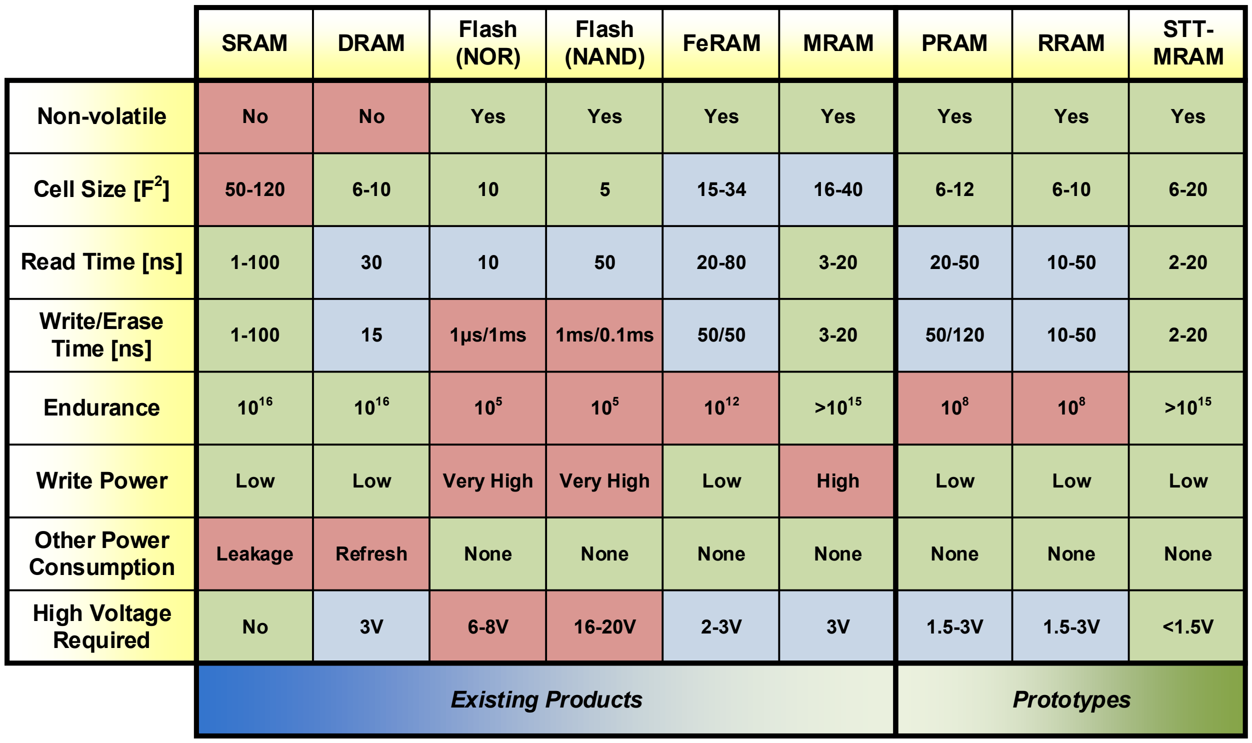
1.1 Sparse-BLAS

With the proliferation of smart phones and tablets over the past several years, there has been an increased demand for mobile devices to support computationally intensive applications, such as augmented reality, neural networks, 3D graphics, portable medical imaging, mobile health monitoring, and various optimization problems–all of which rely on manipulating very sparse data sets [1, 2]. Recently, this persistent challenge of delivering increasing computational complexity in a hand-held form-factor, using a finite power source (i.e. a battery), has been further complicated by the limited or nonexistent improvements in energy efficiency offered by technology scaling. As we scale below 22-nm, energy efficiency by design becomes a necessity for the next generation of mobile system-on-a-chips (SoCs).

The efficient manipulation of sparse data sets is particularly important for the future of compressive sensing (CS) in energy-efficient, mobile, multi-core microprocessors (CPUs) and graphics processing units (GPUs). CS is a technique that exploits the sparsity (in a particular sampling domain) present in most natural signals to simultaneously sample and compress a signal during the data acquisition process. CS allows for data sampling at a much lower frequency, surpassing the traditional limits of Nyquist sampling theory. This has huge implications for traditional communications-based digital signal processing (DSP) applications. Ultra-wide-band communications, spectrum sensing for cognitive radios, and multi-antenna (MIMO) channel estimation are just beginning to incorporate CS into their wireless communications framework to enable low-power, high-data-rate data links [3].

In recent years, tuned software libraries for multi-core microprocessors (CPUs) and graphics processing units (GPUs) have become the status quo for performing sparse linear algebra in high-performance computing (HPC) environments. However, the computational throughput of these libraries for sparse matrices is significantly lower than that of dense matrices, due to a fundamental mismatch between the compression formats required to efficiently store sparse matrices and traditional Von Neumann computing architectures.

This dissertation presents a scalable sparse linear algebra kernel that alleviates the computational and memory bottlenecks present in CPUs and GPUs. Prototyped on an field-



	SRAM	DRAM	Flash (NOR)	Flash (NAND)	FeRAM	MRAM	PRAM	RRAM	STT-MRAM
Non-volatile	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Cell Size [F^2]	50-120	6-10	10	5	15-34	16-40	6-12	6-10	6-20
Read Time [ns]	1-100	30	10	50	20-80	3-20	20-50	10-50	2-20
Write/Erase Time [ns]	1-100	15	1μs/1ms	1ms/0.1ms	50/50	3-20	50/120	10-50	2-20
Endurance	10^{16}	10^{16}	10^{5}	10^{5}	10^{12}	$>10^{15}$	10^{8}	10^{8}	$>10^{15}$
Write Power	Low	Low	Very High	Very High	Low	High	Low	Low	Low
Other Power Consumption	Leakage	Refresh	None	None	None	None	None	None	None
High Voltage Required	No	3V	6-8V	16-20V	2-3V	3V	1.5-3V	1.5-3V	<1.5V
	Existing Products						*Prototypes*		

Figure 1.1: Comparison of memory technologies (source: Wolf et al. [4]).

programmable gate array (FPGA) and implemented as an application specific integrated circuit (ASIC), our proposed architecture is able to increase the energy efficiency of sparse linear algebra by 3 to 4 orders of magnitude over state-of-the-art CPUs and GPUs running the latest software libraries. In the future, integration with spintronics memory devices should improve the energy efficiency by another 1 to 2 orders of magnitude.

1.2 Spintronic Memories

Magnetoresistive Random Access Memories (MRAMs) have attracted a significant amount of interest as a commercially viable universal memory technology. With the density of DRAM, the speed of SRAM, and the non-volatility of Flash it is easy to see why [5] (see Figure 1.1). MRAMs require zero standby power and boast a nearly unlimited programming endurance ($> 10^{15}$ cycles) [6]. Such a memory would eliminate the need for multiple application-specific memories, improving system performance and reliability, while also lowering cost and power

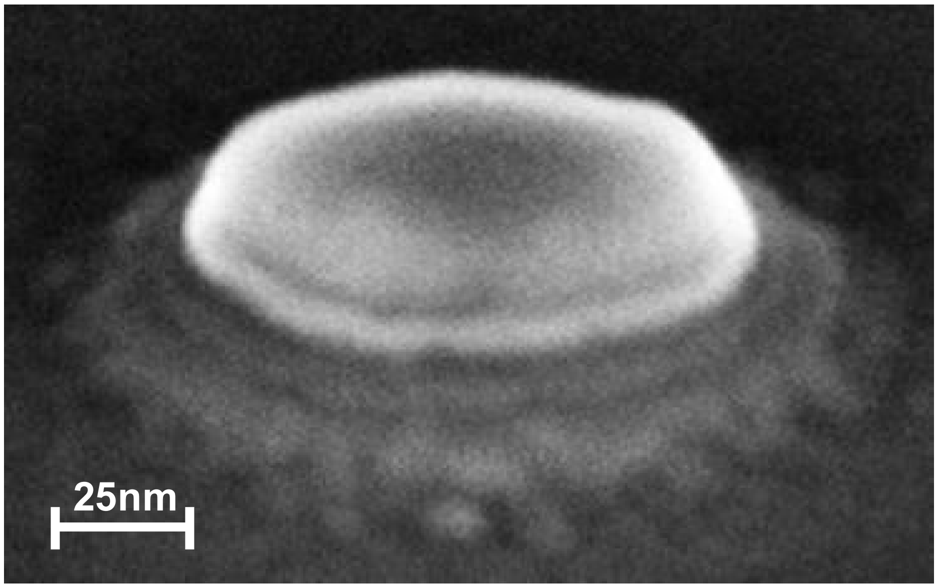


Figure 1.2: SEM photo of an MTJ, courtesy of Mark Lewis.

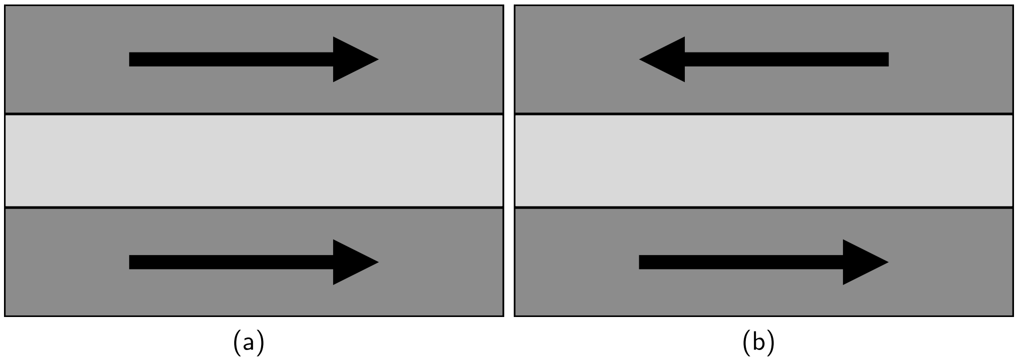


Figure 1.3: MTJ ferromagnetic layers in (a) parallel and (b) antiparallel configurations.

consumption in everything from mobile devices to datacenters [4].

The non-volatile storage element of an MRAM is the Magnetic Tunnel Junction (MTJ). Structurally, an MTJ is a pair of ferromagnets separated by a thin insulating layer. Data storage is achieved by exploiting the magnetic orientation of these ferromagnetic layers [7]. Figure 1.2 shows a scanning electron microscope (SEM) photo of a single MTJ nanopillar. The spintronic operation of the MTJ is discussed later in this work. For now, it is sufficient to understand the MTJ as a pair of ferromagnets separated by a thin insulating layer. Two possible magnetic states arise, the parallel combination of the two layers (Figure 1.3a) and the antiparallel combination (Figure 1.3b).The parallel configuration leads to a low resistive state (R_P), while the antiparallel configuration leads to a high resistive state (R_{AP}) [8].

The discovery of spin-transfer torque (STT) based switching has enabled MRAM to scale below 90nm. Rather than using an indirect current to generate a magnetic field, STT uses a spin-polarized current through the MTJ to accomplish device switching [9]. Toggling of the MTJ is roughly determined by the current density [10]. As the area of the MTJ device decreases, so does the writing current. Spin-Transfer Torque Random Access Memories (STT-RAMs) have the added benefit of being architecturally much simpler than conventional MRAMs [11]. However, with the recent discovery of a voltage-controlled magnetic anisotropy (VCMA) switching mechanism, Magnetoelectric Random Access Memories (MeRAMs) promise to be much dense and more energy efficient that current STT-MRAM technologies. As such, this work discuses the possible integration of both STT-MRAM and MeRAM with our proposed sparse linear algebra kernel.

1.3 Dissertation Outline

Chapter 2 introduces the Basic Linear Algebra subprograms (BLAS) and their sparse-BLAS counterparts. The most common and basic sparse matrix formats are introduced as a prelude to detailing the current state-of-the-art in sparse-BLAS hardware. The final section introduces a variety of sparse matrix algorithms used in bioinformatics.

Chapter 3 takes an in-depth look at the different architectural trade-offs for sparse-BLAS. Key architectural deficiencies in CPU and GPU are first identified. Several strategies are presented for minimizing the number of memory accesses, memory latency, and data hazards to increase the computational efficiency of sparse-BLAS. The chapter concludes with a detailed description of the proposed sparse-BLAS architecture.

Chapter 4 details the development and design of two versions of the proposed architecture on a field programmable gate array (FPGA) system. The first version includes only partial support for sparse-BLAS [12]. The second version expands upon the first to provide full sparse-BLAS compatibility, with the addition of several related scalar functions, capable of executing the sparse bioinformatics algorithms presented in Chapter 2. The basics of reconfigurable computing blocks and prior attempts to use them for sparse-BLAS are also discussed.

Chapter 5 details the implementation of the application specific integrated chip (ASIC) for the proposed algorithm. Details of the processing element, including the data "shuffler," and the memory control scheme for achieving 100% computational efficiency (i.e. utilization) are explained. Design considerations for chip testing and configuration are detailed at the end of the chapter.

Chapter 6 shows the measurement and performance results for both the FPGA implementations and the ASIC implementation for the sparse-BLAS architecture. The FPGA system is able to achieve more than a 50x and 38x improvement in energy efficiency over state-of-the-art CPU and GPU implementations. The ASIC implementation shows a 55x improvement in energy efficiency over our FPGA system, representing more than a 2,750x and 2,085x improvement in energy efficiency over state-of-the-art CPU and GPU implementations, respectively.

Chapter 7 introduced the magnetic tunnel junction (MTJ), and the field of spintronics, specifically focusing on its applications in memories (STT-MRAM and MeRAM). The characteristics and properties of the MTJ device are discussed, with a Verilog-A model capable of capturing these behaviors is presented. The final section of the chapter verifies this model with qualitative and quantitative comparisons to measured devices and detailed micromagentic simulations. The entirety of the source code for the Verilog-A model is presented in Appendix A of this dissertation.

Chapter 8 discusses the design and fabrication results of STT-MRAM and MeRAM in detail. Their integration with CMOS and the sparse-BLAS kernel is also discussed.

Chapter 9 concludes the dissertation and provides some possible directions for future research.

Appendix A contains the Verilog-A code for the MTJ model introduced in Chapter 7.

CHAPTER 2

Sparse Basic Linear Algebra Subprograms

The Basic Linear Algebra Subprograms (BLAS) is a machine-independent, standard application interface developed to help facilitate the portability of linear algebra software across different computer architectures [13]. First published as a Fortran library in 1979, BLAS was later extended to support sparse matrix operations (sparse-BLAS). Sparse matrices arise in a wide variety of computational disciplines, including medical imaging, circuit and economic modeling, industrial engineering, compressive sensing, neural networks, bioinformatics, and algorithms for least squares and eigenvalue problems [14, 15, 16].

Sparse-BLAS operations are categorized into 3 levels [13]. Level 1 operations consist of vector operations: i.e. dot products, vector norms, and vector-vector addition of the form:

$$y =\leftarrow \alpha z + y, \tag{2.1}$$

where y is a dense column vector, z is a sparse column vector, and α is a scalar. Level 2 operations consist of matrix-vector operations, the most important being a general sparse matrix/dense vector multiplication (SpMxV) in the form of:

$$y =\leftarrow \alpha Ax + \beta y, \tag{2.2}$$

where A is a sparse matrix, x is a dense column vector, and β is a scalar. Level 3 operations consist of matrix-matrix operations, the most important being a general sparse matrix/dense matrix multiplication (SpMxM) in the form of:

$$y =\leftarrow \alpha AB + \beta C, \tag{2.3}$$

where B and C are dense matrices. Together, these 3 levels of operations can be used to implement any algorithm involving linear algebra [13].

This chapter is broken up into 4 parts. The first section introduces the simplest and most common sparse matrix storage formats. Section 2 discuss the SpMxV and SpMxM routines

$$A = \begin{bmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 \\ 0 & 0 & 9 & 0 & 6 \end{bmatrix}$$

Figure 2.1: Example sparse matrix A.

$$\begin{aligned} DATA_{COO} &= \begin{bmatrix} 1 & 4 & 2 & 3 & 5 & 7 & 8 & 9 & 6 \end{bmatrix} \\ ROW_{COO} &= \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix} \\ COL_{COO} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 3 & 4 & 2 & 4 \end{bmatrix} \end{aligned}$$

Figure 2.2: COO representation for example matrix A.

in detail. Current state-of-the-art for sparse-BLAS is discussed in Section 3. Section 4 introduces a variety of Bioinformatics algorithms whose performance is closely tied to that of sparse-BLAS.

2.1 Sparse Matrix Representation

There are a variety of ways to represent the sparse matrix for storage purposes. However, the few computationally efficient formats are restricted to highly structured matrices, such as diagonal or banded matrices. As such, our focus is on boosting the computational efficiency of SpMxV and SpMxM for generic sparse matrices. Therefore, we only present general sparse storage schemes in this section. Figure 2.1 shows a sample sparse matrix that will be used to illustrate various sparse matrix storage schemes in the following sections.

$$
\begin{aligned}
DATA_{CSR} &= \begin{bmatrix} 1 & 4 & 2 & 3 & 5 & 7 & 8 & 9 & 6 \end{bmatrix}\\
PTR_{CSR} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}\\
COL_{CSR} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 3 & 4 & 2 & 4 \end{bmatrix}
\end{aligned}
$$

Figure 2.3: CSR representation for example matrix A.

2.1.1 Coordinate List (COO)

The simplest storage scheme, shown in Figure 2.2, is the coordinate (COO) format. The row indices, column indices, and values of the nonzero matrix entries are explicitly stored in 3 separate arrays: *row*, *col*, and *data*. Ideally, the entries are sorted (first by the row index, then by the column index) to improve random access times.

2.1.2 Compressed Sparse Row (CSR)

The compressed sparse row (CSR) format (Figure 2.3) is the most commonly used sparse storage scheme, which also stores the column indices and nonzero values into the arrays: *col* and *data*. Unlike the COO format, the row indices are not explicitly stored, but rather as an array of row pointers, *ptr*. The ith element of *ptr* corresponds to the offset of the ith row into the *col* and *data* arrays. For example, in Figure 2.3 the first element of *ptr* is 0, indicating that the first element in row 0 is 1 and is located in column 0; the second element of *ptr* is 2, indicating that the first element in row 1 is 2 and is located in column 1; the third element of *ptr* is 4, indicating that the first element in row 2 is 5 and is located in column 5; and so on. For an $M \times N$ matrix, *ptr* has $M + 1$ elements in the CSR format, with the final element indicating the total number of nonzero entries in the matrix.

2.1.3 Compressed Sparse Column (CSC)

The compressed sparse column (CSC) format is a variation of the CSR format (Figure 2.4). Instead of storing the column indices and an array of row pointers, the CSC stores the row indices and an array of column pointers. For any matrix A, the CSR storage of A is exactly

$$
\begin{aligned}
DATA_{CSC} &= \begin{bmatrix} 1 & 5 & 4 & 2 & 3 & 9 & 7 & 8 & 6 \end{bmatrix}\\
ROW_{CSC} &= \begin{bmatrix} 0 & 2 & 0 & 1 & 1 & 3 & 2 & 2 & 3 \end{bmatrix}\\
PTR_{CSC} &= \begin{bmatrix} 0 & 2 & 4 & 6 & 7 & 9 \end{bmatrix}
\end{aligned}
$$

Figure 2.4: CSR representation for example matrix A.

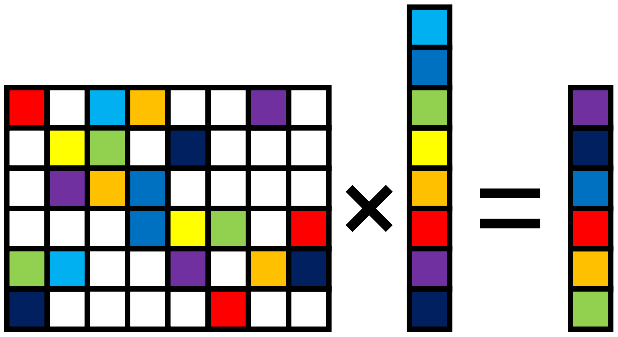


Figure 2.5: A graphical representation of SpMxV.

the same as the CSC storage of A^T.

2.2 Sparse Matrix-Vector and Matrix-Matrix Multiplication

SpMxV is one of the most widely used computational kernel that dominates the performance of many scientific applications [14, 15, 16, 17, 18]. Figure 2.5 shows a graphical representation of SpMxV. SpMxM is a major building block for high-performance graph-based algorithms [19, 20, 21, 22, 23, 23, 24]. Unfortunately, the performance of SpMxV and SpMxM algorithms tends to be much lower than that of dense matrices, mostly due to the mismatch between the memory access patterns of sparse matrices and the compression formats required to efficiently store them [16, 25]. Numerous efforts have been made to accelerate the performance of SpMxV on multi-core microprocessors (CPUs) [1, 26] and graphics processing units (GPUs) over the years [2, 27, 28]. The next section discusses the architectural details of CPUs and GPUs performing sparse-BLAS.

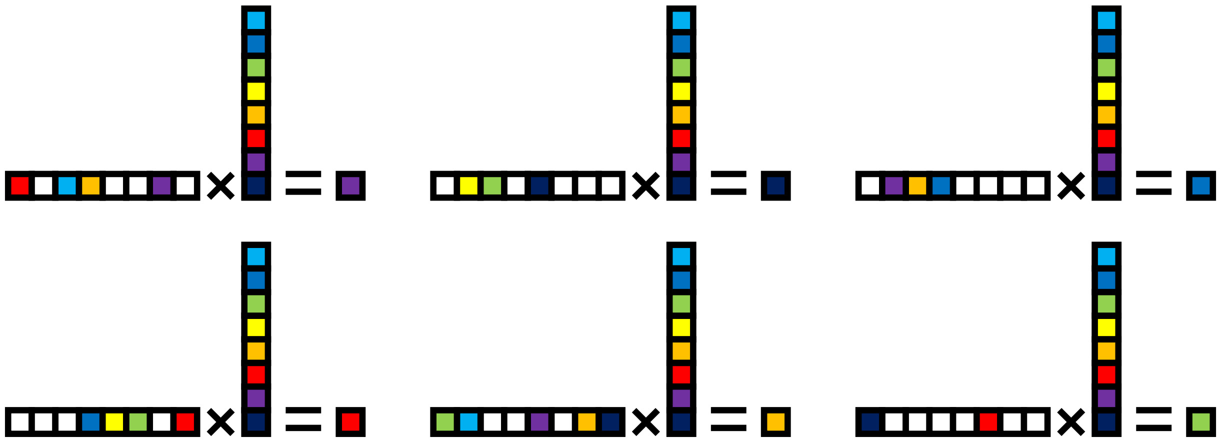


Figure 2.6: A graphical representation of how SpMxV is performed using the CSR format on CPUs and GPUs. Each element in y_i is calculated as the dot product between the ith row of A and the vector x.

2.3 Current State-of-the-Art

One very important metric for gauging the performance and effectiveness of an architecture is its computational efficiency for the target algorithm. Computational efficiency is a measure of the percentage of the total hardware resources available that are actively being used by an algorithm. Computational efficiency correlates very highly with energy-efficiency. Therefore, an implementation of an algorithm with a higher computational efficiency will be the more energy-efficient solution. As such, in this section, we will be using computational efficiency in order to make meaningful comparisons between the performance of different computational architectures.

Specialized software libraries for solving dense and sparse linear algebra problems are very popular for high performance computing. These libraries, such as MKL [26] for CPUs, and cuBLAS [27] and cuSPARSE [28] for GPUs, provide a standardized programming interface, with subroutines optimized for the target platform.

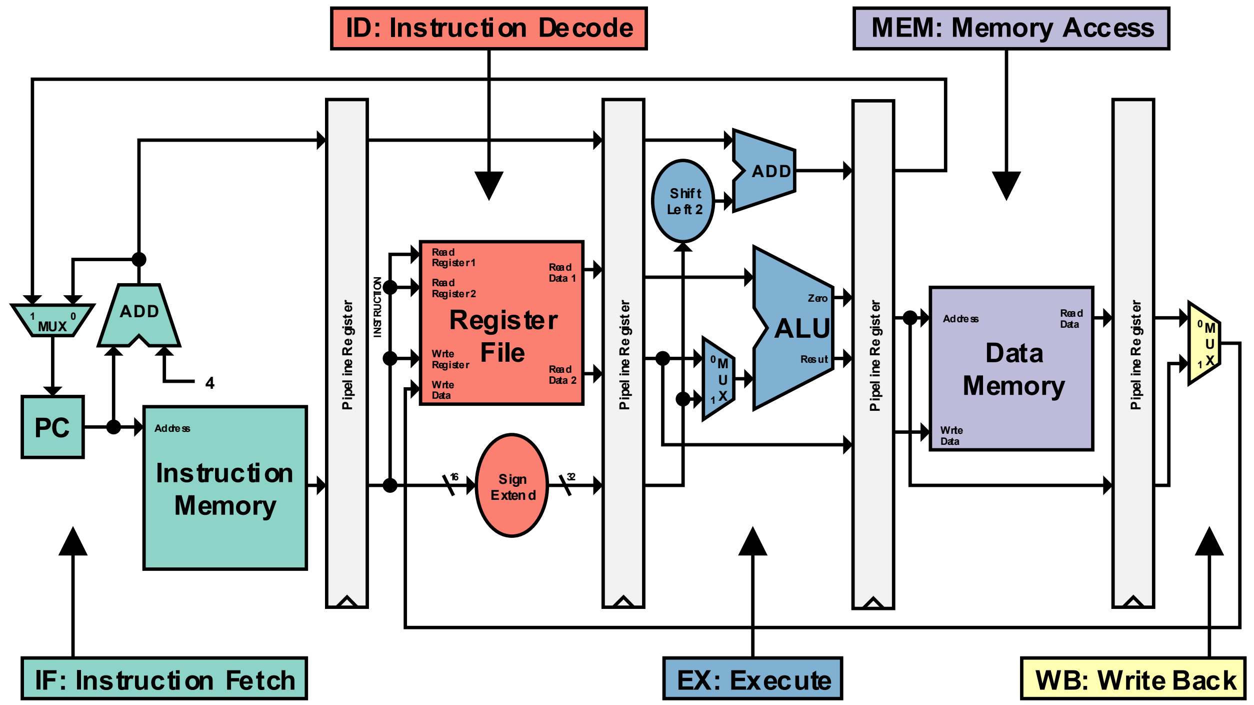


Figure 2.7: A toy example of a 5-stage, pipelined datapath (IF, ID, EX, MEM, WB) for a MIPS microprocessor (based on Figure 6.11 in [29]), with each stage color coded.

2.3.1 Case Study: Efficiency of SpMxV on CPUs

To better illustrate the root cause of computational inefficiencies for performing sparse linear algebra on modern multi-core systems, we will take a detailed look at how modern CPU architectures execute instructions. Figure 2.7 shows a toy example of a 5-stage pipelined datapath in a 32-bit MIPS microprocessor corresponding to the 5 conceptual steps in executing an instruction on a CPU:

(1) Instruction Fetching (IF): Fetch an instruction from (instruction) memory.

(2) Instruction Decoding (ID): Simultaneously decode the instruction and read necessary data from the register file.

(3) Instruction Execution (EX): Execute the specified operation, calculate an address, or pass data through from the register file.

(4) Memory Access (MEM): Read from or write to (data) memory.

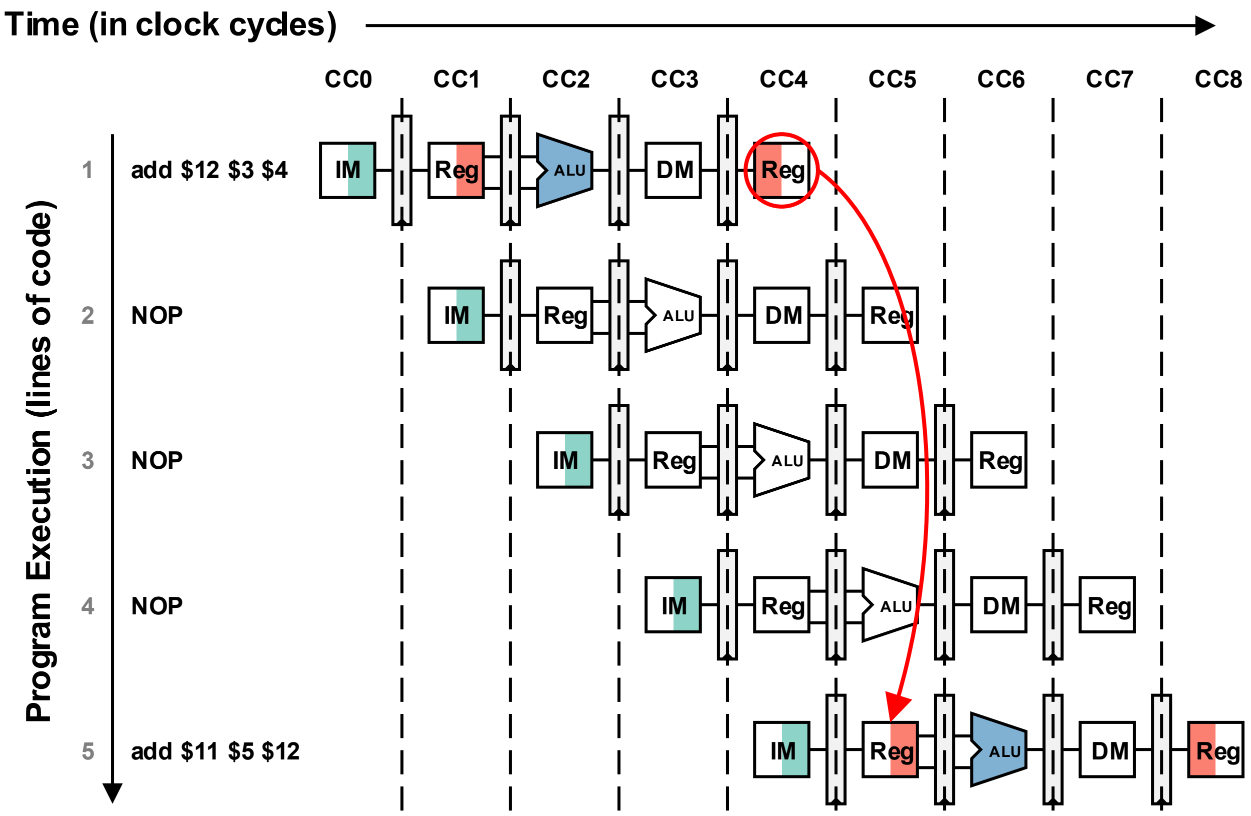


Figure 2.8: Timing for a 5-stage, pipelined datapath demonstrating a stall for a data hazard using NOP (no operation) commands.

(5) Data Write Back (WB): Store the result to the register file.

Pipelining allows multiple instructions to overlap their execution in order to achieve higher performance (i.e. time multiplexing), at the cost of some computational latency. It should also be noted that both the ID and WB stages access the register file at the same time. This can cause a conflict, or hazard, if both the ID and WB steps want to access the same register at the same time. Modern processors handle this situation by forwarding the data to be written from the WB step to the ID step. However, the processor must stall several clock cycles if an instruction wants to access the register file, but, due to the latency of the pipeline, the previous instruction has yet to finish calculating the correct value (see Figure 2.8).

Algorithm 2.1 contains the skeleton code for executing SpMxV using a CSR sparse matrix. When the code from Algorithm 2.1 is compiled for an Intel Core i7 processor, it produces

```c
float btemp;
float *Apos = &A[0][0];
for(int i=0; i<M; i++)
{
  float *xpos = &x[0];
  btemp=0;
  for(int j=0; j<N; j++)
  {
    btemp += (*Apos++) * (*xpos++);
  }
  b[i] = btemp;
}
```

Algorithm 2.1: C pseudo-code for SpMxV.

the x86-64 shown in Algorithm 2.2. By consulting published instruction tables [30] (for instruction latency, throughput, and resource usage) and examining the cache structure of each CPU, we can accurately estimate the number of cycles it will take to perform the SpMxV algorithm. For example, on the i7-2600 processor, line 2 in Algorithm 2.2 takes 2 cycles to execute and must be completed before line 5. However, line 3 can be executed simultaneously with line 2 because they use non-overlapping resources. Similarly, lines 6, 8, 10, 13, and 14 can be executed concurrently with the preceding lines. Line 5 takes one clock cycle, plus the memory fetch latency, to load the value into the floating point unit (FPU). This value needs to finish loading before line 7 can execute. Line 7 takes five clock cycles, plus the memory fetch latency, to load the value into the FPU and multiply it. Line 9 must wait for line 7 to complete, and takes 3 clock cycles to add the values sitting in the FPU. The branching statements in lines 11 and 15 predict a jump and only take two clock cycles, if it is taken. However, if the branch is not taken, a penalty of 18 cycles of latency is incurred to flush the pipeline. Therefore, for an inner loop of length j, it takes:

$$2 \times j \times MEMORY_{latency} + 15 \times j + 23 \tag{2.4}$$

```
loop_i:                 ;
fldz                    ; btemp = 0
mov    eax, hXXXX       ; j = N
loop_j:                 ;
fld    dword ptr [edx]  ; A_ij
add    edx, 4           ; Apos++
fmul   dword ptr [ecx]  ; A_ij*x_j
add    ecx, 4           ; xpos++
faddp  st(1), st        ; btemp = btemp+A_ij*x_j
dec    eax              ; j--
jnz    short loop_j     ; loop if j!=0
fst    dword [ebx]      ; b_i = btemp
add    ebx, 4           ; bpos++
dec    esi              ; i--
jnz    short loop_i     ; loop if i!=0
```

Algorithm 2.2: x86-64 pseudo-code for SpMxV.

clock cycles to execute $2 \times j$ floating-point operations in the SpMxV algorithm. Ideally, the FPU can perform one floating-point operation per clock cycle. Dividing the ideal $2\times j$ number of clock cycles by the number of clock cycles in Equation 2.4 results in a computational efficiency of:

$$1\Big/\left(MEMORY_{latency} + \frac{15 \times j + 23}{2 \times j}\right). \tag{2.5}$$

For a very large A matrix (i.e. $\gg$ cache size), due to the random, non-sequential memory accesses for the CSR format, the L1 and L2 caches will frequently miss and hit either the L3 cache or DRAM. Therefore, the average memory access latency of the algorithm will be roughly equal to that of the L3 cache, or 28 clock cycles for a i7-2600 processor [31]. Furthermore, if we assume a range of approximately 1 to 100 nonzero elements per row, we can estimate the i7-2600 processor's computational efficiency to be between 2.12% and 3.46% for SpMxV. This poor efficiency is due to the SpMxV kernel spending roughly 77.6% of the time waiting for memory fetches, 18.5% of the time waiting for the FPU to calculate an intermediary result (i.e.stalling to resolve a data hazard), and 1.9% of the time executing loops and other control flow. A first-order estimation of the computational efficiency, using a 1-to-1 FLOP to memory fetch ratio and a memory latency of 28, for SpMxV on the i7-2600

```
mov     reg1, hXXXX    ; (j)   reg1 = M
mov     reg2, h0000    ; (b_i) reg2 = 0
mov     reg3, hXXXX    ; (A_ij ptr)
mov     reg4, hXXXX    ; (X_j loc ptr)
loop:                  ;
load    reg3 into reg5 ; (A_ij ptr) reg5 = A_ij
load    reg4 into reg6 ; (X_j loc) reg6=loc X_j
add     reg3, 4        ; (A_ij ptr) ++
add     reg4, 4        ; (X_j loc ptr) ++
load    reg6 into reg7 ; (X_j ptr) reg7 = X_j
fma     reg2 reg5 reg7 ; b_i = b_i + A_ij*x_j
dec     reg1           ; j--
jnz     reg1 loop      ; loop if j~=0
```

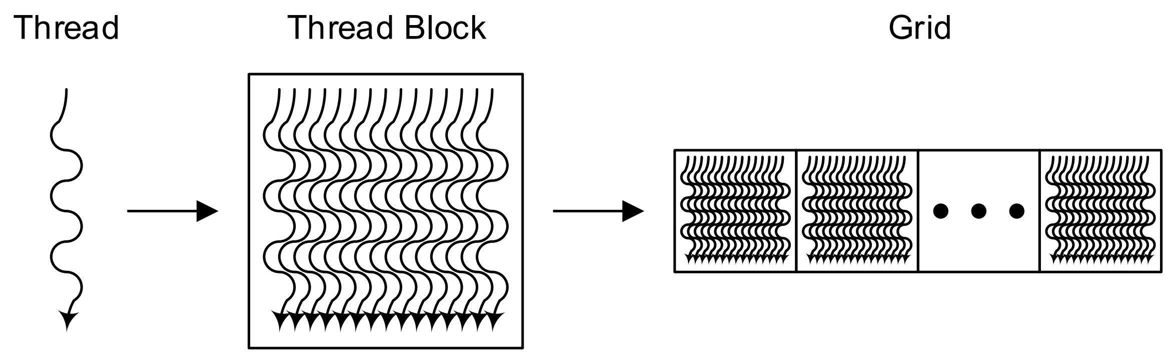
Algorithm 2.3: CUDA-like pseudo-code for SpMxV.

processor is 3.44%, which is in good agreement with our previous calculations.

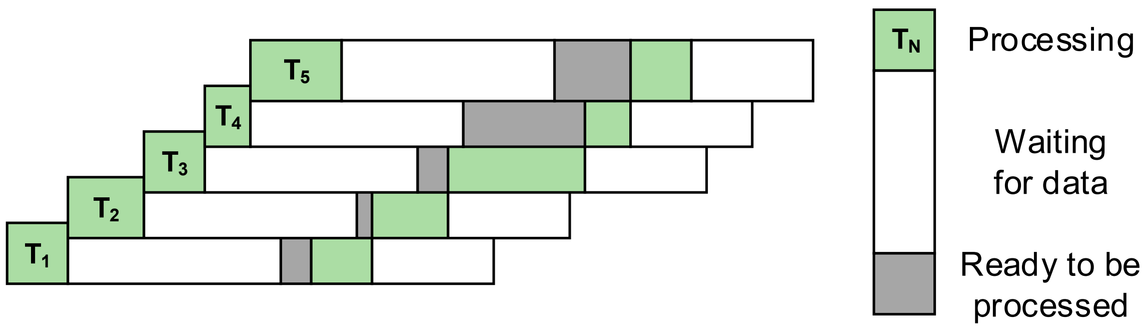
2.3.2 Case Study: Efficiency of SpMxV on GPUs

GPUs process data in a very different fashion from CPUs—which have a handful of FPUs that operate independently of each other. GPUs typically have hundreds or thousands of FPUs that process data in a single-instruction/multiple-data (SIMD) fashion. In a CUDA-enabled GPU, processes are broken up into threads, which are organized into warps or thread blocks (see Figure 2.9(a)). These thread blocks are further grouped into grids, which are executed on streaming multiprocessors (a collection of FPUs executing the same instruction on different threads). Each CUDA core (FPU) in the streaming multiprocessors continues to execute a single thread until either it finishes, requires a memory fetch, or encounters a hazard. In a CPU, each of these events would necessitate a pipeline stall. However, in a CUDA-enabled GPU, the CUDA core will instead switch to a new thread waiting in the queue to be processed (see Figure 2.9(b)). By doing this, a CUDA-enabled GPU is usually able to hide the large latencies associated with memory accesses and branching statements which are on the order of hundreds of clock cycles.

When the code from Algorithm 2.1 is compiled for an NVIDIA CUDA-enabled GPU,



(a)



(b)

Figure 2.9: The (a) organization and (b) execution of multiple threads on a CUDA-enabled NVIDIA GPU.

it produces the pseudo-CUDA code shown in Algorithm 2.3. For a large A matrix, with a large number of threads dedicated to calculating the SpMxV, we can ignore the latency of the arithmetic/operation pipeline due to thread switching. Therefore, lines 1-4, 8, 9, 12, and 13, in Algorithm 2.3, take only one clock cycle each to execute. Line 11 takes an average of 1.2 cycles to execute, because only 160 of the 192 CUDA cores can perform floating-point math on a streaming multiprocessor [32]. Since we are accessing memory sequentially, lines 6 and 7 take roughly a single L1 cache latency to load. This latency cannot be masked by thread switching, because every other thread will also be trying to access memory. Line 10 takes approximately a single L3 latency to load its value do to the random, non-sequential nature of its access pattern. For a GTX TITAN GPU, the L1 and L3 latencies are 15 and 215 clock cycles, respectively [32]. This results in a computational efficiency of 0.40%, with the kernel spending roughly 98% of the time waiting for memory fetches, 0.1% of the time waiting for the FPU to calculate an intermediary result, and 1.5% of the time executing loops and other control flow. A first-order estimation of the computational efficiency, using a 1-to-1 FLOP to memory fetch ratio and a memory latency of 215, for SpMxV on the GTX TITAN GPU is 0.46%. From this we can deduce that GPUs are roughly 6 times less computationally efficient than CPUs for memory-bounded algorithms, like the SpMxV kernel, precisely because off-chip memory latencies are roughly 6 times longer in GPUs than CPUs.

2.4 Sparse Matrix Algorithms in Bioinformatics

Sparse matrix operations are the dominant computational kernel for most graph-based and compressive sensing algorithms. The two most common operations are the SpMxV,

$$y \leftarrow \alpha A x + \beta y, \tag{2.6}$$

and SpMxM,

$$Y \leftarrow \alpha A X + \beta Y, \tag{2.7}$$

where α and β are scalars, A is an $M \times N$ sparse matrix, x and y are $N \times 1$ (sparse or dense) vectors, X is an $N \times K$ (sparse or dense) matrix, and Y is an $M \times K$ (sparse or dense) matrix.

This section reviews several algorithms for building, searching, and clustering interactomes, as well as high resolution tomographic imaging, whose computation is dominated by SpMxV and SpMxM. We also introduce an algorithm complexity metric corresponding to the number of FLOP per memory access. If we know the ratio of FLOP per memory fetch for an algorithm and the underlying hardware architecture it is being executed on, we can estimate the computational efficiency of the system. For example, an algorithm with a 10-to-1 FLOP to memory access ratio, executing on a system with memory latency of 2 and a logic latency of 1, would have an expected computational efficiency of $100\% \times 10/(10+2) = 83.3\%$.

2.4.1 Markov Clustering Algorithm

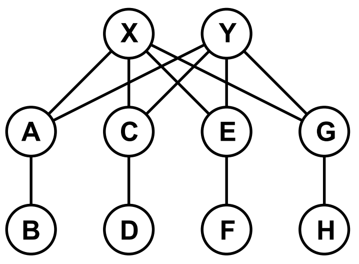
SpMxM forms the core of the Markov Clustering algorithm (MCL), an important bioinformatics algorithm for determining cluster information in graph networks [19, 21]. MCL has been applied to a wide range of interactomes, such as gene-regulatory and protein-protein interaction networks [22, 23]. MCL is not only fast, but also more robust to noise and graph uncertainty than other clustering algorithms [23, 24].

The MCL algorithm preforms unsupervised clustering by simulating the process of random walks (or flow) within a graph via the alteration of two operations: *expansion* and *inflation* [21]. In a graph theory, a *natural cluster* is a group of nodes that contain many edges between members of the cluster. Additionally, the shortest path (number of edges) between nodes of the cluster and any arbitrary node not in the cluster should be high. In other words, a random walk of a graph will infrequently move from one natural cluster to another.

These random walks are achieved using the column stochastic (Markov) matrix M associated with the graph G. M is defined by normalizing all columns of the adjacency matrix (A) corresponding to G (see Figure 2.10(a)-(d)). Expansion corresponds to computing the expected location of an observer after p steps of a random walk within the graph. To do this, the MCL **expansion** operator takes the pth power of the square matrix $M \in \mathbb{R}^{n\times n}$ [21]:

$$\mathrm{Exp}_p M = M^p. \tag{2.8}$$

Inflation seeks to strengthen the probability of intra-cluster walks and weaken inter-cluster walks by taking the Hadamard power of M, following by a normalization step to maintain column stochasticity, to create a new Markov matrix. Given a matrix $M \in \mathbb{R}^{m\times n}$, $M \geq 0$ and a number $r \in \mathbb{R}$, $r > 0$, the **inflation** operator Γ_r ($\mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$) of M, with power



(a)

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

(b)

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

(c)

$$\begin{bmatrix} 0.25 & 0.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0.2 \\ 0.25 & 0.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.25 & 0.5 & 0 & 0 & 0 & 0 & 0.2 & 0.2 \\ 0 & 0 & 0.25 & 0.5 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.25 & 0.5 & 0 & 0 & 0.2 & 0.2 \\ 0 & 0 & 0 & 0 & 0.25 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0.5 & 0.2 & 0.2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0.5 & 0 & 0 \\ 0.25 & 0 & 0.25 & 0 & 0.25 & 0 & 0.25 & 0 & 0.2 & 0 \\ 0.25 & 0 & 0.25 & 0 & 0.25 & 0 & 0.25 & 0 & 0 & 0.2 \end{bmatrix}$$

(d)

$$\begin{bmatrix} 0.48 & 0.45 & 0.06 & 0 & 0.06 & 0 & 0.06 & 0 & 0.06 & 0.06 \\ 0.20 & 0.45 & 0 & 0 & 0 & 0 & 0 & 0 & 0.02 & 0.02 \\ 0.06 & 0 & 0.48 & 0.45 & 0.06 & 0 & 0.06 & 0 & 0.06 & 0.06 \\ 0 & 0 & 0.20 & 0.45 & 0 & 0 & 0 & 0 & 0.02 & 0.02 \\ 0.06 & 0 & 0.06 & 0 & 0.48 & 0.45 & 0.06 & 0 & 0.06 & 0.06 \\ 0 & 0 & 0 & 0 & 0.20 & 0.45 & 0 & 0 & 0.02 & 0.02 \\ 0.06 & 0 & 0.06 & 0 & 0.06 & 0 & 0.48 & 0.45 & 0.06 & 0.06 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.20 & 0.45 & 0.02 & 0.02 \\ 0.07 & 0.05 & 0.07 & 0.05 & 0.07 & 0.05 & 0.07 & 0.05 & 0.40 & 0.28 \\ 0.07 & 0.05 & 0.07 & 0.05 & 0.07 & 0.05 & 0.07 & 0.05 & 0.28 & 0.40 \end{bmatrix}$$

(e) $chaos = 0.647$

$$\begin{bmatrix} 0.60 & 0.63 & 0.03 & 0 & 0.03 & 0 & 0.03 & 0 & 0.04 & 0.04 \\ 0.19 & 0.31 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.03 & 0 & 0.60 & 0.63 & 0.03 & 0 & 0.03 & 0 & 0.04 & 0.04 \\ 0 & 0 & 0.19 & 0.31 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.03 & 0 & 0.03 & 0 & 0.60 & 0.63 & 0.03 & 0 & 0.04 & 0.04 \\ 0 & 0 & 0 & 0 & 0.19 & 0.31 & 0 & 0 & 0 & 0 \\ 0.03 & 0 & 0.03 & 0 & 0.03 & 0 & 0.60 & 0.63 & 0.04 & 0.04 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.19 & 0.31 & 0 & 0 \\ 0.06 & 0.03 & 0.06 & 0.03 & 0.06 & 0.03 & 0.06 & 0.03 & 0.45 & 0.39 \\ 0.06 & 0.03 & 0.06 & 0.03 & 0.06 & 0.03 & 0.06 & 0.03 & 0.39 & 0.45 \end{bmatrix}$$

(f) $chaos = 0.472$

$$\begin{bmatrix} 0.83 & 0.85 & 0 & 0 & 0 & 0 & 0 & 0 & 0.02 & 0.02 \\ 0.11 & 0.13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.83 & 0.85 & 0 & 0 & 0 & 0 & 0.02 & 0.02 \\ 0 & 0 & 0.11 & 0.13 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.83 & 0.85 & 0 & 0 & 0.02 & 0.02 \\ 0 & 0 & 0 & 0 & 0.11 & 0.13 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.83 & 0.85 & 0.02 & 0.02 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.11 & 0.13 & 0 & 0 \\ 0.03 & 0.01 & 0.03 & 0.01 & 0.03 & 0.01 & 0.03 & 0.01 & 0.46 & 0.46 \\ 0.03 & 0.01 & 0.03 & 0.01 & 0.03 & 0.01 & 0.03 & 0.01 & 0.46 & 0.46 \end{bmatrix}$$

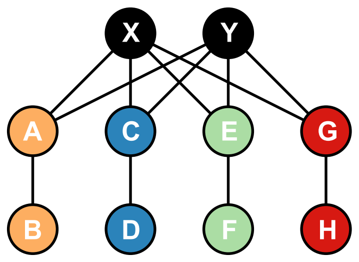
(g) $chaos = 0.204$

$$\begin{bmatrix} 0.97 & 0.98 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.03 & 0.02 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.97 & 0.98 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.03 & 0.02 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.97 & 0.98 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.03 & 0.02 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.97 & 0.98 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.03 & 0.02 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0.5 \end{bmatrix}$$

(h) $chaos = 0.029$

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0.5 \end{bmatrix}$$

(i) $chaos = 1.83 \times 10^{-8}$



(j)

Figure 2.10: The Markov Clustering algorithm for (a) an example graph G with 10 nodes. The corresponding (b) adjacency matrix of G (Step 1)and it's modification to (c) include self loops for convergence (Step 2). Normalizing the all of the columns produces the (d) Markov matrix M (Step 3). The result of the (e) first and (i) final iteration of the expansion and inflation operations with their corresponding chaos (Steps 4-6). And finally, the example graph G with (j) the identified clusters highlighted (Step 7).

coefficient r, is defined by [21]:

$$(\Gamma_r M)_{ij} = (M_{ij})^r \Bigg/ \sum_{k=1}^{m} (M_{kj})^r \ , \tag{2.9}$$

for $i = 1 \dots m$ and $j = 1 \dots n$. $\Gamma_r M$ is the inflation matrix of M with a power coefficient r. Iterating the expansion and inflation processes results in a idempotent matrix, with the steady-state separation of the graph into different segments. Since no paths (or edges) exist between these segments, they are interpreted as clusters. The steady-state condition is numerically achieved when the global chaos, according to Equation 2.10 and Equation 2.11, of the kth column of matrix M in the current iteration is less then a minimum threshold value e, for all k [19]:

$$\text{chaos}_k = \frac{\max((\Gamma_r M)_{ik})}{\sum_{i=1}^{m} (\Gamma_r M)_{ik}^2} \tag{2.10}$$

for $i = 1 \dots m$ and

$$\text{glb\_chaos} = \max(\text{chaos}_k) \tag{2.11}$$

for $k = 1 \dots n$.

To summarize the MCL algorithm for a graph G, an expansion power p, an inflation parameter r, and a chaos threshold of e, the steps are:

Step 1: Create an Adjacency matrix A corresponding to the graph G.

Step 2: (optional) Add self loops to each node of A.

Step 3: Normalize the columns of A to produce the Markov matrix M.

Step 4: Expand M by taking the pth power.

Step 5: Inflate the resulting matrix from step 4 using parameter r.

Step 6: Repeat steps 4 and 5 until a steady state is reached (i.e. glb_chaos $\leq e$).

Step 7: Interpret the resulting matrix to discover clusters.

An example of the MCL algorithm for simple graph is given in Figure 2.10. For N_{nz} nonzeros in M, the SpMxM of the expansion operation requires at most $2pN_{nz}N$ FLOP and at least $pN_{nz}N$ memory fetches, a 2-to-1 FLOP to memory access ratio. Together, the inflation and the chaos calculation require $8N_{nz}$ FLOP and N_{nz} memory fetches, an 8-to-1 FLOP to memory access ratio. The computational complexity of the expansion operation clearly dominates the performance of the MCL algorithm. However, as we will show, MCL has a much higher ratio of FLOP per memory access than other sparse algorithms.

2.4.2 PageRank Algorithm

PageRank is a link analysis algorithm that assigns a numerical weight to each node in a graph G denoting its relative importance within the set. Originally developed to rank the relative importance of web pages for Internet searches [34], the PageRank algorithm has become an increasingly popular tool for investigating the complex interactions present in large biological networks [35,36] (see Figure 2.11). The algorithm's stability in the presence of a large number of false positive and false negative interaction edges makes it very attractive for protein-protein interaction networks.

The algorithm assumes a random surfer model [34]: i.e. a person will continue to randomly click web links with a probability d and will randomly teleport to another web page with a probability $(1-d)/N$, where N is the total number of web pages. The PageRank of each node at time t is represented by the vector $R(t) \in \mathbb{R}^{n\times 1}$ with the probability of clicking each link represented by the stochastic (Markov) matrix $M \in \mathbb{R}^{n\times n}$ (constructed in the same manner as MCL). The PageRank algorithm iterates:

$$R(t+1) = dMR(t) + \frac{1-d}{N}v, \tag{2.12}$$

where v is a column vector of all ones, until some minimum difference e:

$$|R(t+1) - R(t)| < e, \tag{2.13}$$

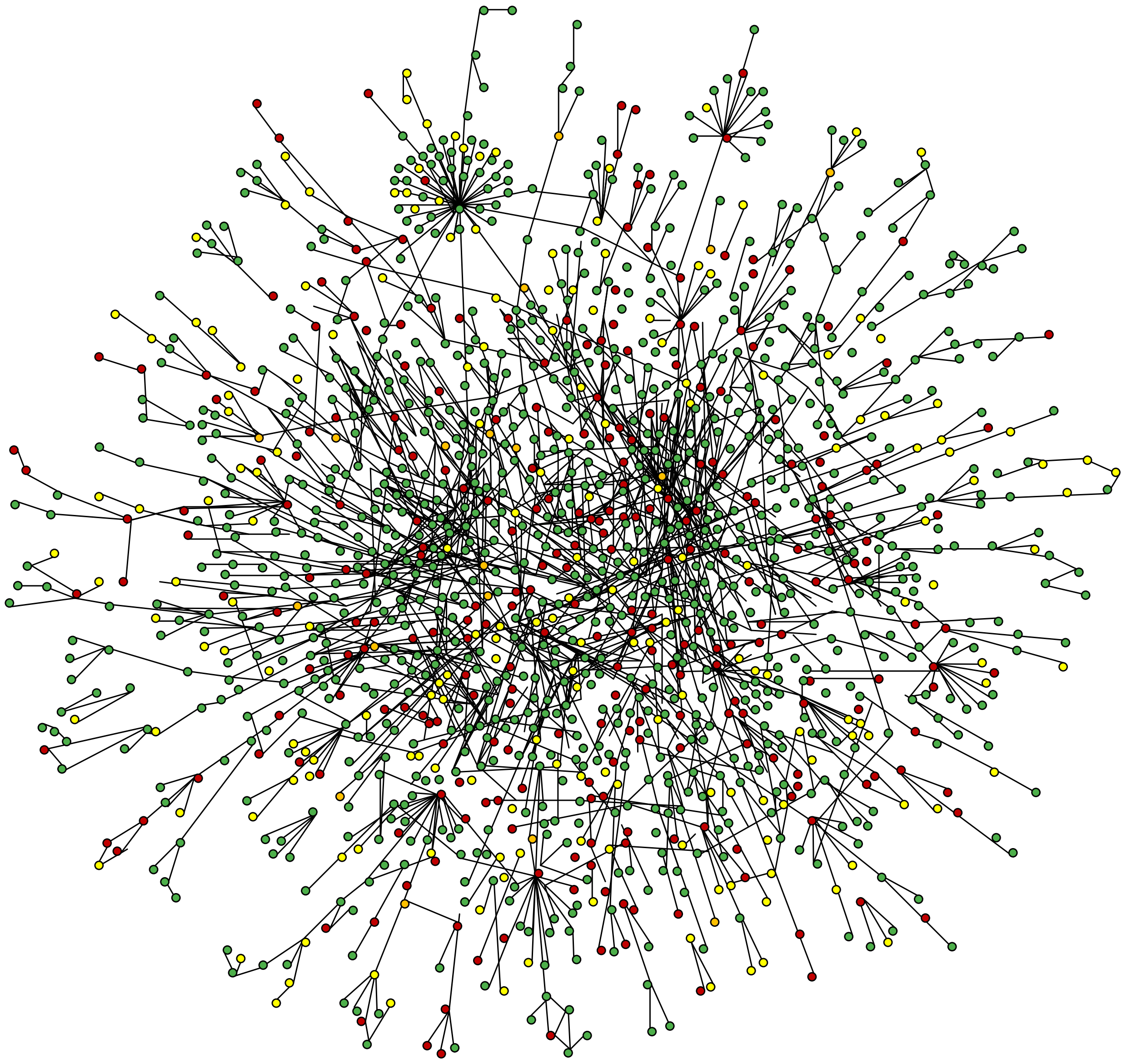


Figure 2.11: Visualization of the network of 2,358 protein-protein interactions for the yeast *Saccharomyces cerevisiae* reported in [33]

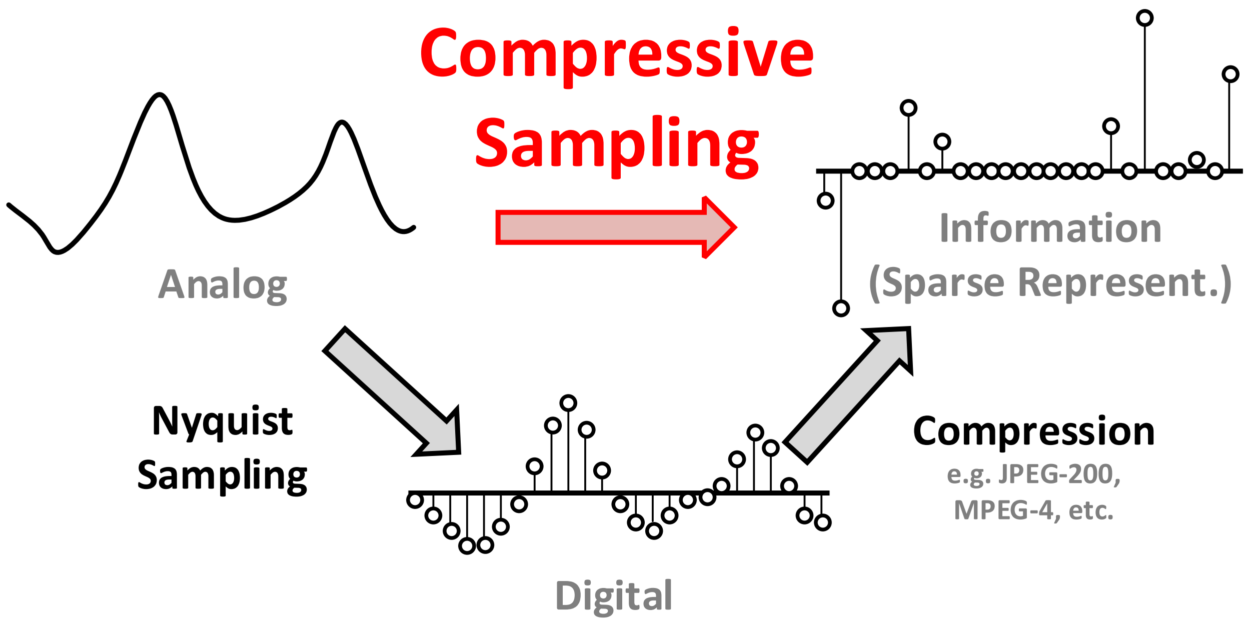


Figure 2.12: Diagram demonstrating the concept of compressive sensing.

is satisfied between successive iterations. The computation of each of these iterations is dominated by SpMxV. For N_{nz} nonzeros in M, each iteration requires $2N_{nz} + 2N$ FLOP and $2N_{nz} + 2N$ memory fetches, a 1-to-1 FLOP to memory access ratio.

The concept of a *Personalized PageRank* can obtained by modifying the column vector v to be 1 for nodes of interest and 0 otherwise [34]. For $M < N$ nodes of interest, the Personalized PageRank PR is computed as:

$$PR(t+1) = dMPR(t) + \frac{1-d}{M}v, \tag{2.14}$$

with the same stopping conditions. The computational complexity of Personalized PageRank is exactly the same as PageRank.

2.4.3 Compressive Sensing

Compressive Sensing (CS) is a signal processing technique in sampling theory for the efficient acquisition and recovery of sparse signals by finding solutions to underdetermined linear systems [37]. CS has many applications in statistics, information theory, theoretical computer science, various engineering disciplines, and bioinfomatics where it has found its niche in group testing and DNA microarrays [38, 39] for DNA-DNA hybridization experiments [40], studying gene expression [41, 42], and building gene interaction networks [43].

In CS theory, a signal x that is k-sparse (i.e. a signal with n elements and $k \ll n$ nonzero elements) can be perfectly reconstructed from $m \geq k \log(n)$ measurements via [37]:

$$\min_{y=\tilde{x}\Phi} \|\tilde{x}\|_{\ell_1}, \tag{2.15}$$

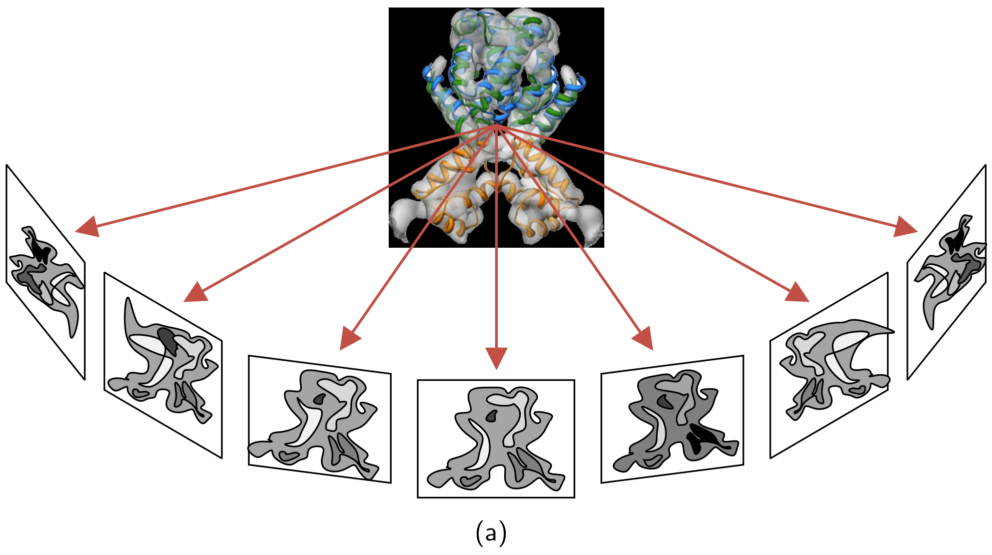
where $\tilde{x}$ is the reconstructed signal and $\Phi \in \mathbb{R}^{m \times n}$ is the sampling matrix. Additionally, for a random sparse Φ, $\tilde{x}$ can be iteratively reconstructed via SpMxV in linear time ($O(k)$) with, roughly, a 5-to-3 FLOP to memory access ratio [44].

2.4.4 Electron Tomography

Electron Tomography is a commonly used technique in the field of structural biology to obtain detailed 3D images of complex biological structures at molecular resolutions by combining 2D images taken from various angles [45, 46]. Iterative reconstruction techniques, like the Simultaneous Iterative Reconstruction Technique (SIRT), preform very well when handling incomplete and noisy data sets for high resolution imaging [47]. However, the poor computational efficiency of the SpMxV kernel in SIRT leads to long reconstruction times and has limited its adoption.

For a 2D image $X \in \mathbb{R}^{n \times n}$ with $B \in \mathbb{R}^{\Theta \times d}$ projection measurements, where Θ is the number of projection angles and d is the number of detectors, and $x = vec(X) \in \mathbb{R}^{n^2 \times 1}$ and $b = vec(B) \in \mathbb{R}^{\theta d \times 1}$, where x and b are obtained by stacking the columns of X and B respectively, the data acquisition process $A \in \mathbb{R}^{\theta d \times n^2}$ is given by [48]:

$$Ax = b. \tag{2.16}$$


(a)

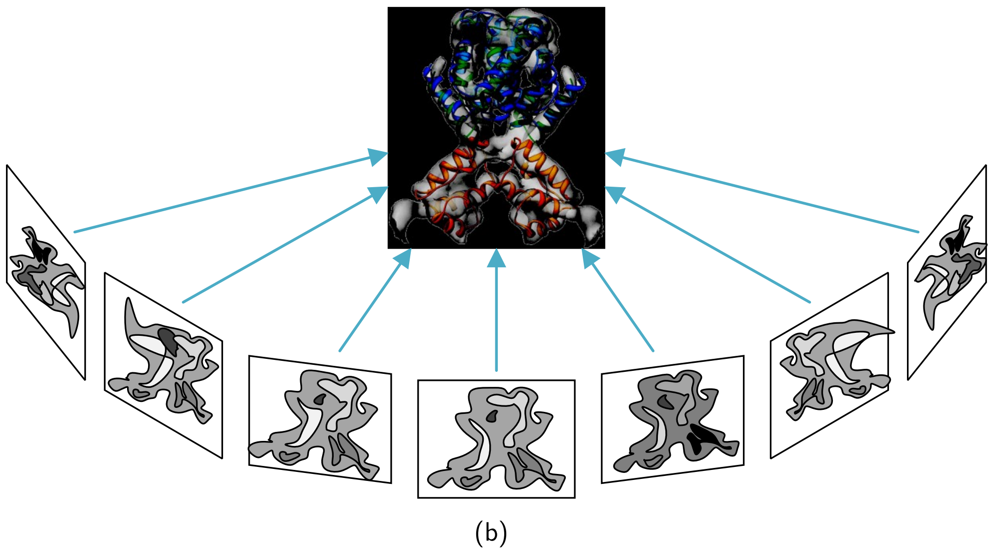

(b)

Figure 2.13: A diagram demonstrating the (a) acquisition and (b) reconstruction of a subcellular structures for 3D Electron Tomography.

The SIRT reconstruction algorithm then iterates [48]:

$$x_{k+1} = x_k - \lambda A^T \left(A x_k - b \right), \tag{2.17}$$

where λ is a scalar and x_k is the kth estimation of x, for a predefined number of cycles. 3D image reconstruction can be obtained by substituting $X \in \mathbb{R}^{n \times n \times n}$ and $B \in \mathbb{R}^{\Theta \times d \times d}$. For all practical tomographic recording processes, A is a very sparse matrix and every iteration of

SIRT requires two SpMxV. For N_{nz} nonzeros in A, each iteration of SIRT (for a 2D image) requires $4N_{nz}+2n^2+\Theta d$ FLOP and a minimum of $4N_{nz}+2n^2+2\Theta d$ memory fetches, less than a 1-to-1 FLOP to memory access ratio. As such, SIRT is an extremely memory-bounded algorithm.

CHAPTER 3

A Scalable Architecture for Sparse Linear Algebra

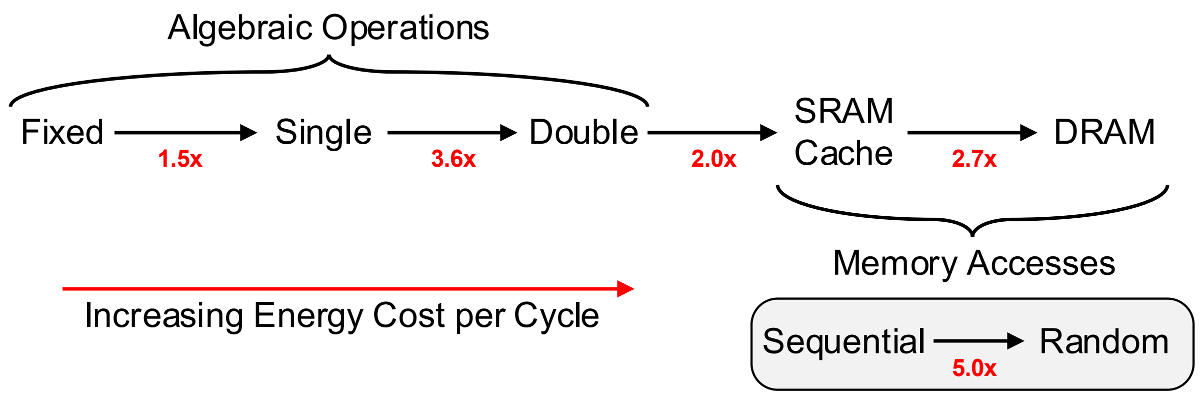


Figure 3.1: Energy cost per clock cycle for algebraic and memory operations.

This chapter takes an in-depth look at the different architectural trade-offs for sparse-BLAS. The first section delves into energy efficiency by quantifying the relationship between the amount of energy per logical and memory operations. The second section takes a look at the architectural trade-offs for sparse-BLAS and identifies the key deficiencies in CPU and GPU limiting their energy efficiency. Several strategies are presented for minimizing the number of memory accesses, memory latency, and data hazards to increase the computational efficiency of sparse-BLAS. The final section concludes the chapter with a detailed description of the proposed sparse-BLAS architecture.

3.1 Energy Efficiency

Energy efficiency is the most important issue facing the field of HPC today [49]. The exponential growth in computational performance of super computers has had a corresponding exponential growth in their electrical usage. The world's largest super computers require tens of megawatts to power and cool them, at a cost of millions of dollars a year to operate [50]. The high financial cost of energy delivery and cooling ultimately limits the scalability and peak performance of many HPC systems. For example, a single 42U server rack, common to most university labs, still requires tens of kilowatts to power and many thousands of dollars a year to operate [50]. Improving the energy efficient of HPC systems would increase their scalability and peak performance, while simultaneously reducing their operation costs.

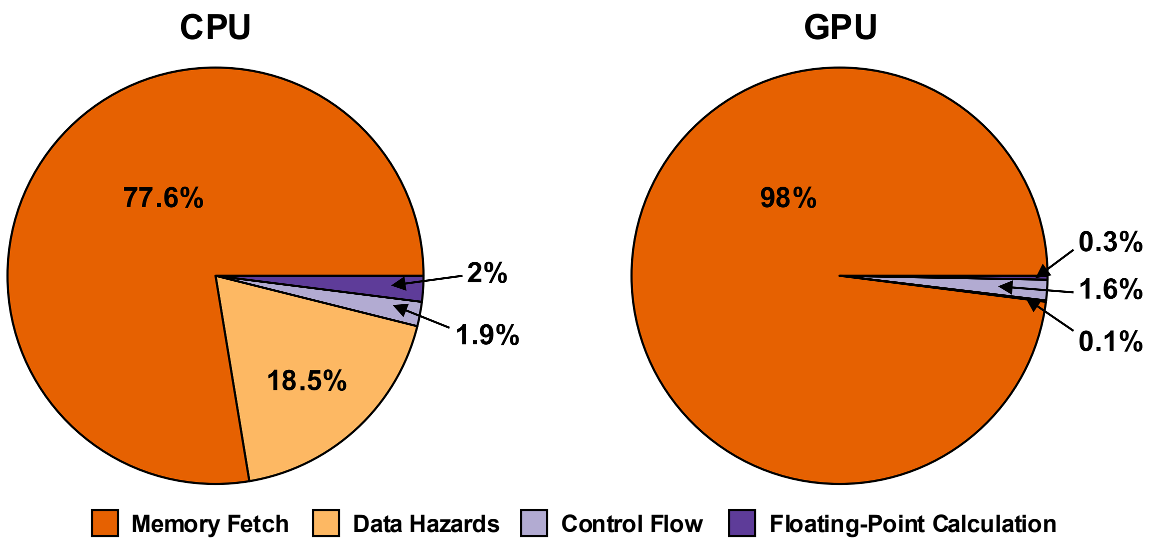


Figure 3.2: Computational efficiency breakdown for a CPU and GPU. Only the floating-point calculations are considered "useful" work. Memory fetches, data hazards, and program control flow are viewed as wasted clock cycles.

From Figure 3.1, we can see that memory accesses are very costly; 15-30x less energy efficient that fixed-point arithmetic [51, 52, 53, 54]. Their energy efficiency is even worse if we must access memory randomly, rather than in a sequential fashion [54]. In a modern cell phone, more than half of the power burned is from static (standby) power dissipation [55]. Following sections look at these architectural trade-offs for sparse-BLAS.

3.2 Architectural Trade-offs for Sparse-BLAS

This section focuses on the architectural trade-offs for sparse-BLAS. Key architectural deficiencies in CPU and GPU are first identified. We then take a look at several strategies to reduce memory accesses, data hazards, and optimizing memory sizes.

3.2.1 Identifying Key CPU/GPU Architectural Inefficiencies

For SpMxV (and SpMxM) on CPUs and GPUs, y_i ($Y_{i,j}$) is almost exclusively calculated as the dot product of the ith row of A and the vector x (jth column of $X = X_j$). This is because each computing core usually contains only a handful of general purpose registers and a single floating-point unit (FPU). Additionally, off-chip memory accesses incur a large latency penalty, typically dozens of clock cycles for CPUs and hundreds of clock cycles for GPUs. Using the CSC format would require more off-chip memory accesses than the CSR format, due to the limited number of registers. Because of this, CSR is one of the most computationally efficient storage options for sparse matrices on CPUs and GPUs. The CSR format has the added benefit of being easily parallelizable: each computing core can be independently assigned a different value of y_i ($Y_{i,j}$) to calculate. Unfortunately, SpMxV and SpMxM have several drawbacks on CPUs and GPUs that hurts their overall computational efficiency [16]:

(1) The SpMxV and SpMxM kernels are memory-bounded. Additionally, CPUs and GPUs typically have much larger computational throughput than available memory bandwidth. This leads to a very low utilization rate for the computing resources, and subsequently, poor energy efficiency.

(2) The indirect (global) memory references for the vector x (X_j) present in *col* adds uncertainty to the memory access pattern, ultimately delaying the computation via unmaskable latency. Each element of *col* must first be loaded from memory and added to the address of x (X_j) as an offset. Only then can the correct value of x (X_j) be loaded into the FPU for computation.

(3) Irregular memory access of vector x (X_j) causes a large number of cache misses. In CPUs, this cache miss can add tens of cycles of latency. In GPUs, a cache miss can add hundreds of cycles of latency. GPUs typically try to hide these large latencies by interleaving dozens of threads on a single computational core. This works well for computation-bounded algorithms, but not memory-bounded algorithms like those with SpMxV and SpMxM. As a result, random memory accesses are 5 times less energy efficient than sequential memory accesses [54].

(4) Short row lengths (i.e. very few nonzero elements per row) can cause serious performance degradation. When rows are short, the overhead associated with calculating

each element of y (Y) becomes significant.

Due to these drawbacks, CPUs and GPUs reach less than 5% of their theoretical peak processing throughput and utilize less than 50% of their available memory bandwidth for sparse linear algebra [1,2].

3.2.2 Minimizing Memory Accesses

The optimization of memory accesses is critical for improving the computational efficiency of sparse linear algebra. However, as we saw in the previous section, CPUs and GPUs are limited to CSR matrix storage formats (and their derivatives) due to hardware architecture limitations. Additionally, from Chapter 2, we saw that the maximum attainable performance of the row-wise dot product between A and x (Equation 2.2) for the CSR format is dictated by the worst-case system memory latency. In an attempt to reduce the number of memory accesses, the proposed sparse-BLAS kernel instead relies on the CSC matrix format to calculate SpMxV and SpMxM as a series of column-wise vector additions of A weighted by each element of x. Fundamentally, this allows us to directly address the major limitations present in current CPU and GPU systems:

(1) The architecture allows for much better balancing of system resources. The number of processing elements (PEs) and operating frequency can be tailored to match the available memory bandwidth to optimize energy efficiency.

(2) What used to be indirect (global) memory references for x in the *col* vector (for the CSR format) are now direct (local) memory references for y in the *row* vector. In other words, when a column of A is weighted by an element of x, we know exactly which elements of y the partial products will contribute to. This allows us to almost halve the number of required memory accesses, one of the largest bottlenecks in the SpMxV and SpMxM kernels.

(3) Memory access to the x vector is no longer irregular, but sequential. By using the CSC format to store A, both A and x can be placed in a large off-chip memory and

sequentially streamed into the co-processor (eliminating the time and energy overheads of a cache miss present in CPUs and GPUs).

(4) Short row or column lengths have much less impact on the performance, since the PEs are rarely idled thanks to the balanced memory bandwidth and computing capability. However, the performance can be degraded if the memory bandwidth of x approaches that of A, as in the case of extremely sparse matrices (nonzero densities <0.0001%). In the rare case of $M \gg N$, the performance of CSC is no better than that of CSR due to the significant increase in the memory bandwidth required by x.

A simple proof of claim (2), that we can in most cases halve the number of memory accesses, is as follows: The number of memory accesses required for x and A using the CSR format is $2 \times N_{Nz}$. For the CSC format it is $(1 + N_{Nz} \text{ per Column}) \times N_{column} = N_{Nz} + N_{column}$. Taking the ratio of CSC to CSR memory accesses we get:

$$\frac{1}{2} + \frac{N_{column}}{2 \times N_{Nz}}. \tag{3.1}$$

Using the matrices from Table 6.1 in Chapter 6, used to benchmark the proposed architecture, we find ratio from Equation 3.1 to be between 0.50 and 0.66, with an average of 0.54. In the extreme case that $N_{column} = N_{Nz}$, then the ratio if Equation 3.1 is equal to 1. thus confirming claim (4).

3.2.3 Strategies to Reduce Data Hazards

One strategy to reduce the number of data hazards is data stream reordering. This is not possible with the CSR data format, forcing us to stall the processor whenever a data hazard occurs. An example of this stalling behavior (for the CSC format) is shown in Figure 3.3 for a single processing element performing SpMxV between the example A matrix from Figure 2.1 and the vector $x = [0.1\ \ 0.2\ \ 0.3\ \ 0.4\ \ 0.5]^T$. In the example, the latency of the floating-point multiplier and adder are both 2 clock cycles. Additionally, the simple dual-port memory

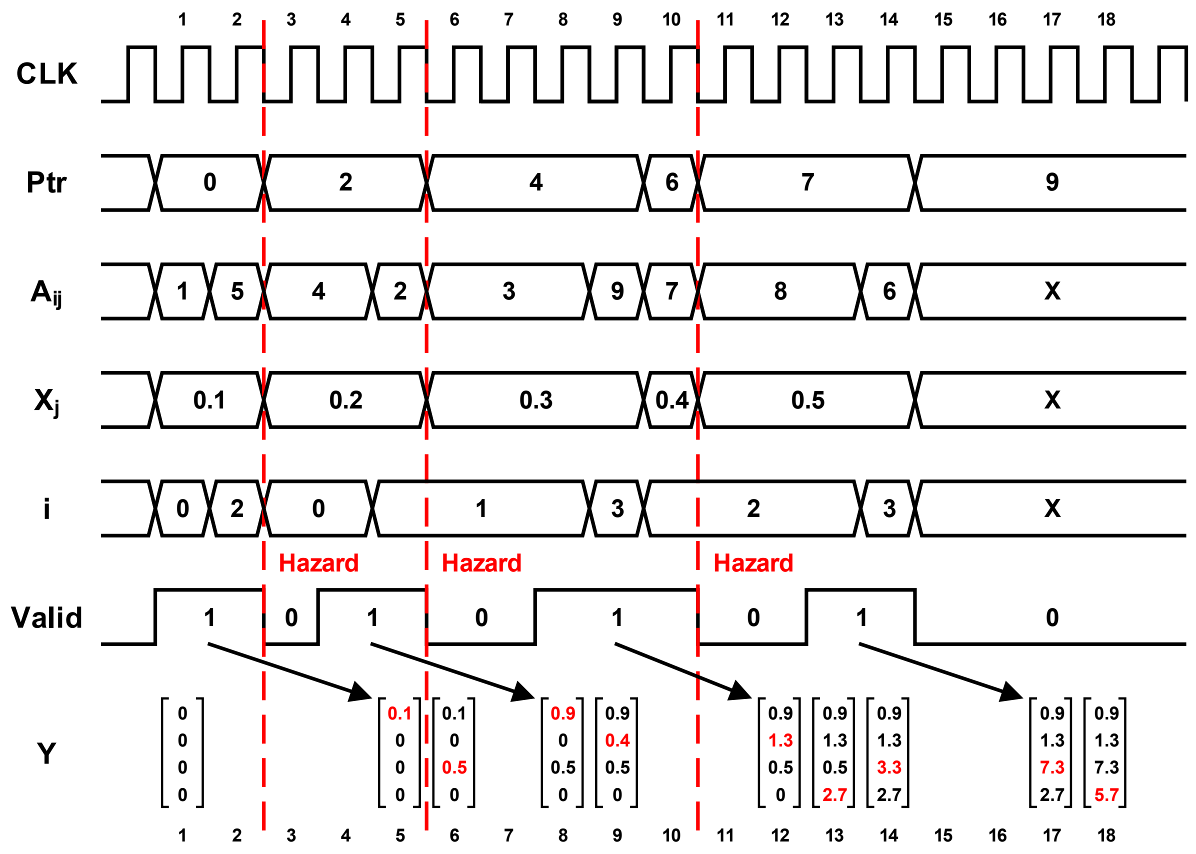


Figure 3.3: Timing diagram for calculating the SpMxV of the example A matrix from Figure 2.1 and $x = [0.1\ \ 0.2\ \ 0.3\ \ 0.4\ \ 0.5]^T$ using a single PE, without data stream reordering. For this example, the floating-point adder and multiplier both have a latency of 2 clock cycles and the simple dual-port memory (Y) has a latency of 1 clock cycle.

(Y) has a latency of 1 clock cycle. In cycle 2, a hazard is detected due to the proximity of the partial product of 1×0.1 and the partial product of 4×0.2. We must stall for 1 cycle—by deasserting *Valid* and holding the values of A_{ij}, X_jk, and i—to ensure that partial product of 1×0.1 is added to Y before continuing. If the co-processor had not stalled, the partial product of 4×0.2 would have added itself to the current value of Y, 0, producing an incorrect final value of 0.8 (instead of 0.9). Computation resumes in cycle 4, after the hazard has passed. Additional hazards are detected in cycles 5 and 10, with each hazard resulting in 2 clock cycles of stalling.

However, with the CSC data format, we aren't forced to operate on the same row. This allows us to switch processing to an alternate row whenever a data hazard occurs. Figure

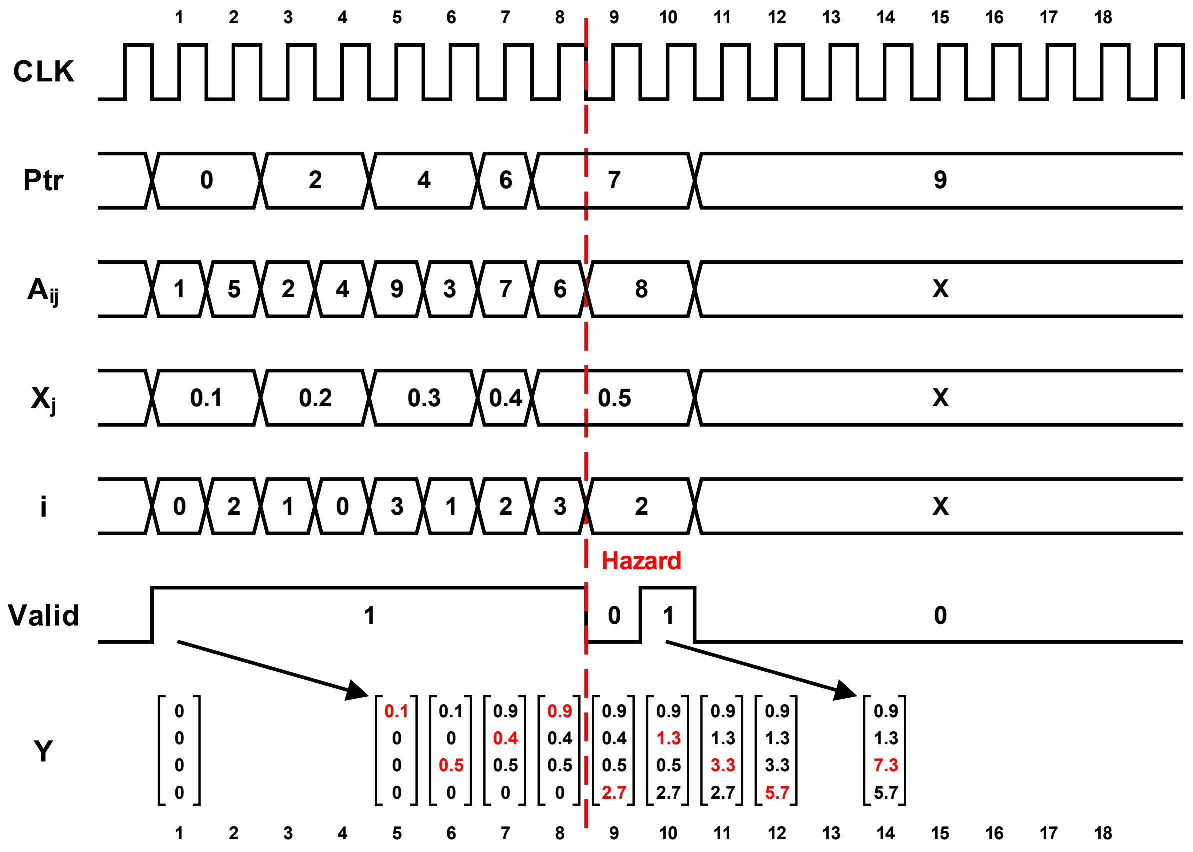


Figure 3.4: Timing diagram for calculating the SpMxV of the example A matrix from Figure 2.1 and $x = [0.1 \;\; 0.2 \;\; 0.3 \;\; 0.4 \;\; 0.5]^T$ using a single PE, with data stream reordering. For this example, the floating-point adder and multiplier both have a latency of 2 clock cycles and the simple dual-port memory (Y) has a latency of 1 clock cycle.

3.4 shows an example of this behavior for a single processing element performing SpMxV between the example A matrix from Figure 2.1 and the vector $x = [0.1 \;\; 0.2 \;\; 0.3 \;\; 0.4 \;\; 0.5]^T$. In the example, the latency of the floating-point multiplier and adder are both 2 clock cycles. Additionally, the simple dual-port memory (Y) has a latency of 1 clock cycle. In cycle 2, a hazard is detected due to the proximity of the partial product of 1×0.1 and the partial product of 4×0.2. Instead of stalling, the processor swaps in the partial product of 2×0.2, pushing the partial product of 4×0.2 back one clock cycle. In cycle 4, another hazard is detected due to the proximity of partial product 2×0.2 and the partial product of 3×0.3. Again instead of stalling, the partial product of 9×0.3 is substituted, pushing back the calculation of the partial product of 3×0.3 by one clock cycle. Finally, in cycle 8, another

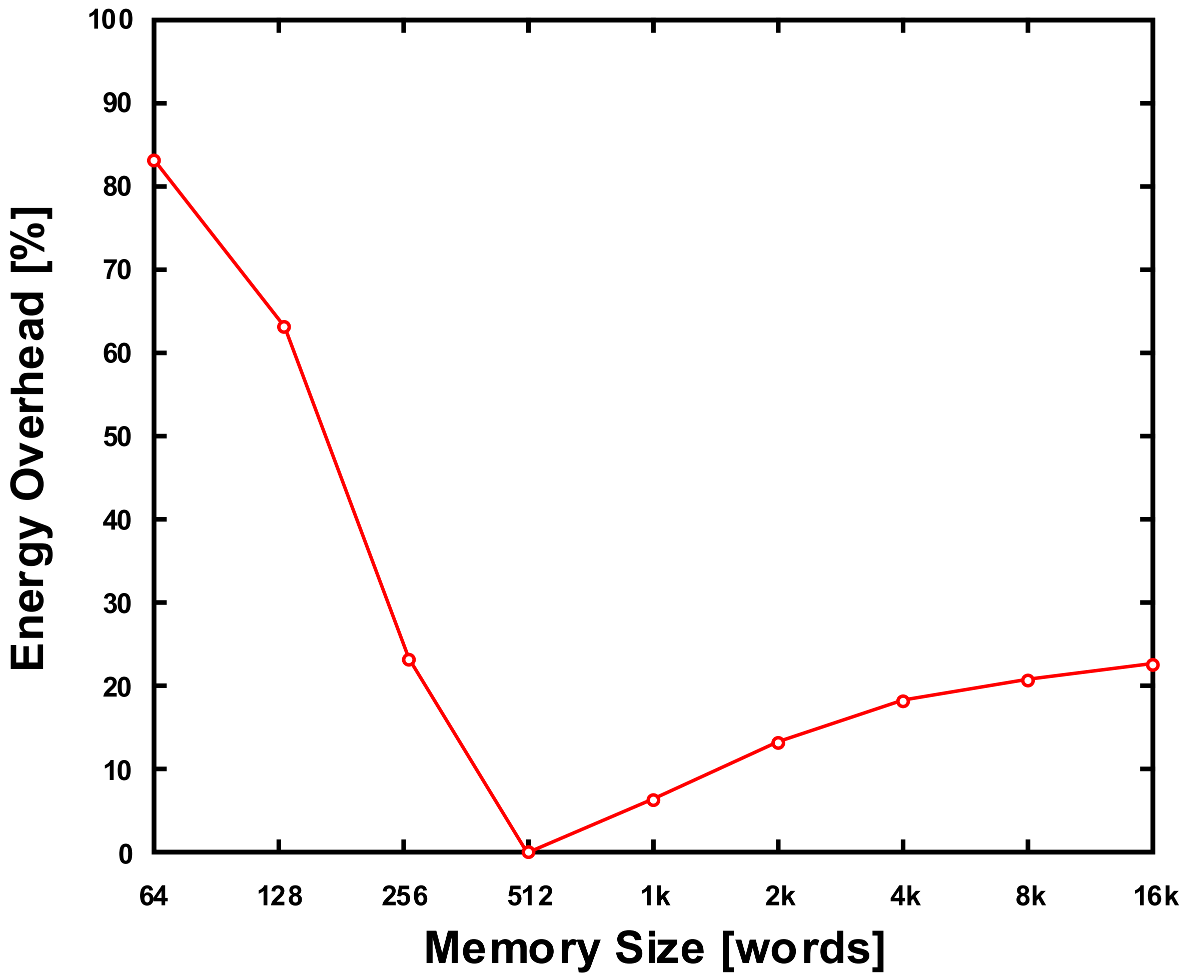


Figure 3.5: Bandwidth overhead vs. memory size.

hazard is detected due to the proximity of partial product 7×0.4 and the partial product of 8×0.5. However, since we have reached the end of the data stream, we are forced to stall the processor one clock cycle to allow the data hazard to resolve itself.

The simple examples in Figure 3.3 and Figure 3.4 demonstrate an 23% speedup for calculating SpMxV through the use of data stream reordering.

3.2.4 Memory Size vs. Energy Efficiency

It should be noted that Equation 3.1 assumes that the system has a large enough memory to fit the entirety of the y vector in it. However, this does not necessarily hold true for a

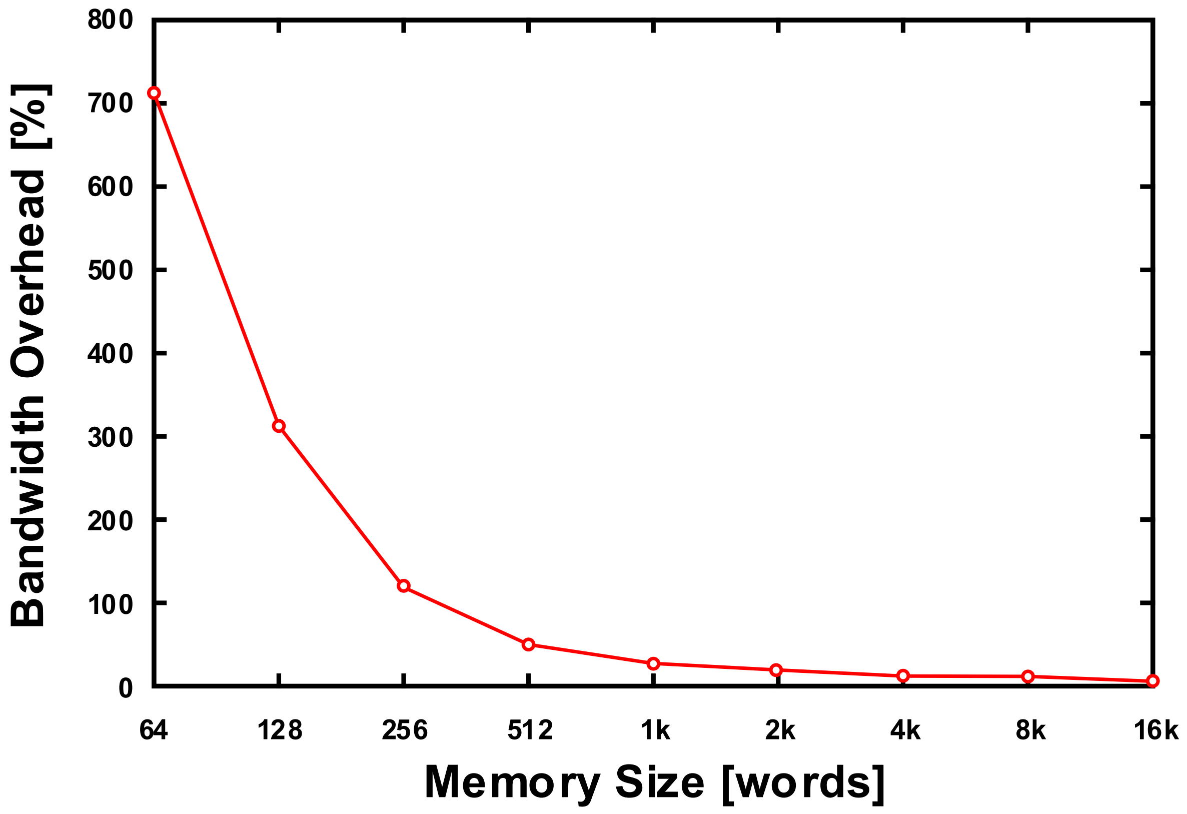


Figure 3.6: Energy overhead vs memory size.

finite memory size. If the y vector is larger than the memory we are using, then we will need to partition the A matrix into blocks to accommodate the smaller memory. The result is an increase in the required memory bandwidth, which is detrimental to bandwidth limited mathematical kernels like sparse-BLAS. Therefore, it is necessary to carefully optimize the size of our memories to limit this degradation.

Figure 3.5 shows the average memory bandwidth overhead for various memory sizes, when compared to the ideal case, for the matrices from Table 6.1 in Chapter 6. Assuming that this overhead is directly proportional to the execution time of sparse-BLAS and that the leakage energy of a memory is directly proportional to it's size, then we can calculate the energy efficiency loss (or overhead) for a particular memory size. The results of this calculation for the matrices from Table 6.1 are shown in Figure 3.6, which has been normalized to the optimal memory size (512 words).

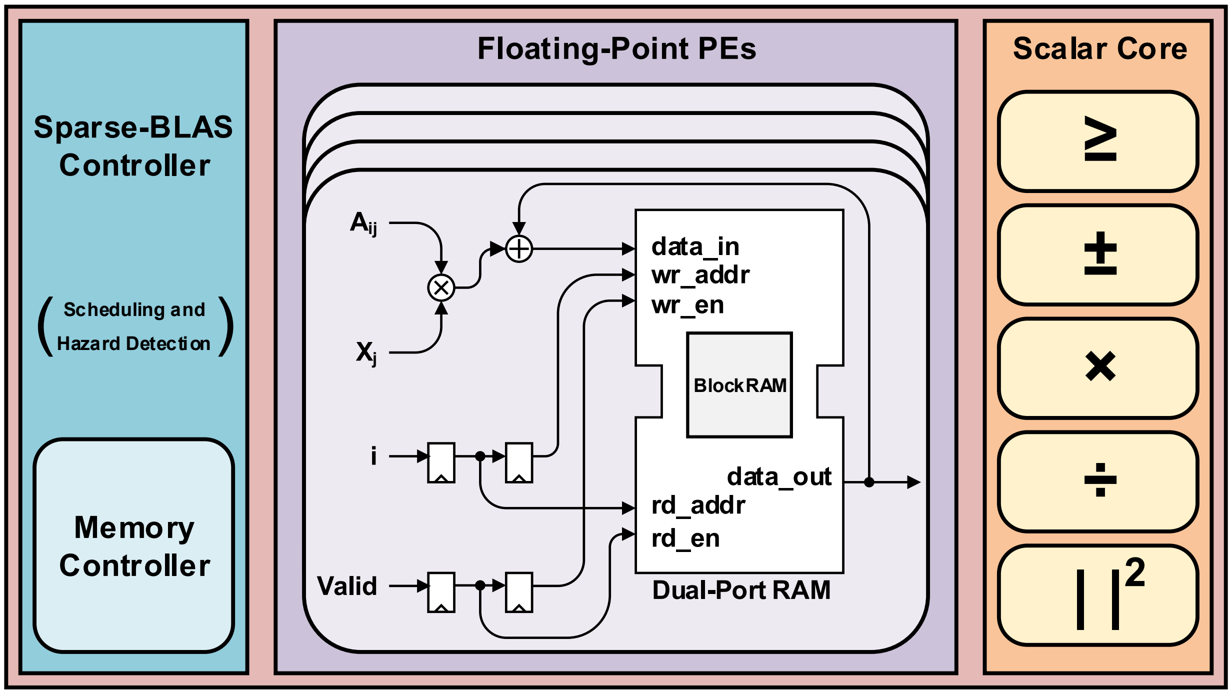


Figure 3.7: Top-level schematic of the proposed sparse-BLAS kernel.

3.3 Proposed Architecture

Figure 3.7 shows the top level schematic of the proposed sparse-BLAS kernel. It consists of a sparse-BLAS controller, with an integrated memory controller, various processing elements for the CSC data format, and an optional scalar core to support non-BLAS math operations for executing more complex algorithms. The following sections with look into the details of each of these blocks.

3.3.1 Processing Element

Each processing element (PE) contains a single Floating-Point Unit, as well as a simple dual-port RAM, and a data-stream reorderer (or "Shuffler"). To perform sparse-BLAS, each PE multiplies an element of the data vector (A_{ij}) and the corresponding element of the x vector (x_j) together. The resulting partial product is then added to the address in the row vector (i), before being stored back into the dual-port RAM. Due to the latency of the multiplication and addition operations, a *Valid* signal is used to prevent data corruption

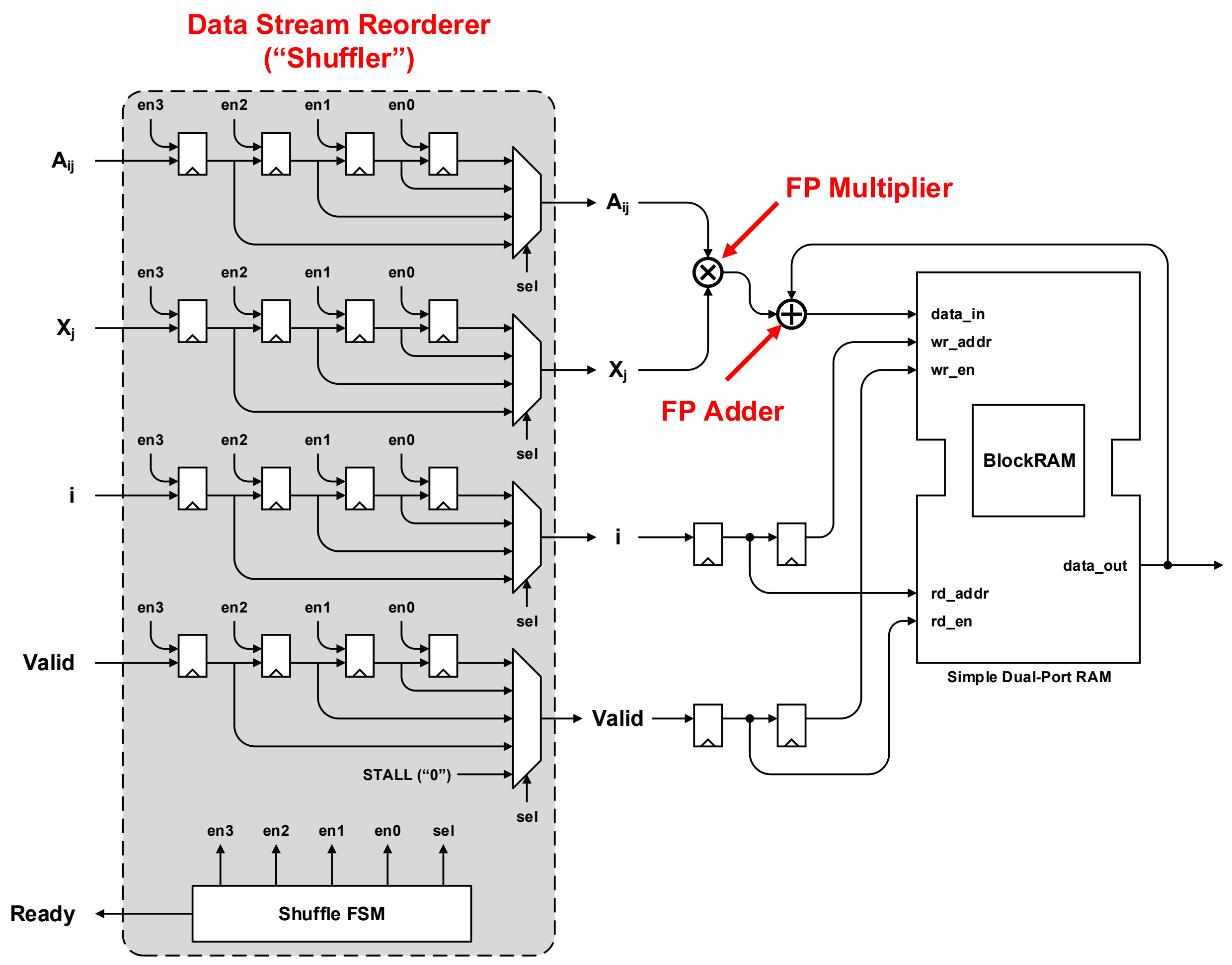


Figure 3.8: Block diagram of the proposed processing element with a floating-point unit, dual-port memory, and data "shuffler."

due to hazards. The "Shuffler" attempts to reduce the number of data hazards by first monitoring for them, and then swapping in alternative partial products (instead of stalling) until the data hazard is resolved. Using this strategy, data can be continuously streamed into each PE (directly from the CSC format) with a small startup overhead equal to the combined latency of the adder, multiplier, and "Shuffler".

3.3.1.1 Floating-Point Unit

The Floating-Point Unit (FPU) of each PE contains a floating-point adder and multiplier. The purpose of the FPU is to perform the partial product multiplication and accumulation for each element of the data vector (A_{ij}) and the corresponding element of the x vector (x_j). In this work, we use the single-precision floating point format (binary32) specified by the IEEE 754-2008 standard [56]. We use the default rounding mode: round to nearest, ties to even.

3.3.1.2 Dual-Port Memory

Since each PE can only store a finite number of elements, each PE will need to contain a partial working copy of the vector being computed. To facilitate this, blocking is performed along the rows of A (i.e. each PE is assigned a subset of rows of A for computation). The final vector is then assembled by concatenating the output of each PE during memory write-back (requiring no additional latency). This parallelization strategy (essentially a block CSC matrix storage format) preserves the property of sequential memory accesses across all PEs to ensure high computational efficiency.

To ensure that we can both compute and update the partial products in the same clock cycle, a dual-ported memory is required. The first port of the memory is set to a read-only state to provide data for the partial product accumulation. This port is also used to write out the contents of the dual-ported memory to main system memory. The second port is fixed into a write only state, with a write enable signal, to write back the updated partial products. This port can also be used to zero-out the memory, or load a predefined vector in the most general case of SpMxV and SpMxM.

3.3.1.3 The "Shuffler"

The data-stream reordering unit, or "Shuffler, consists of a buffer for N items each of A_{ij}, X_j, i, and $Valid$ data used to calculate a partial product, and a FSM to control the flow of data in the buffer. The FSM monitors each piece of data in the buffer to see if it will create a data hazard further down the pipeline if it is issued to the FPU. Every clock cycle the FPU requests a new piece of data. The FSM will issue the first element in the buffer that will not create a data hazard. If the buffer is empty or if every valid piece of data in the buffer would create a data hazard, the FSM will issue a stall command to the FPU. An example of this type of behavior can be seen in Figure 3.4.

3.3.2 Sparse-BLAS Controller

The primary purposes of the sparse-BLAS controller are scheduling and hazard detection. The following sections will discuss data flow and memory management of the controller.

3.3.2.1 Memory Management

The memory controller acts as a slave to the sparse-BLAS controller, ensuring a continuous stream of data into the PEs. As we have see before, data hazards arise when two or more partial products want to write to the same memory address of the dual-port RAM in the PE in a short period of time. Due to the latency of the floating-point adder of the PE, shown in Figure 3.8, the existing sum of the partial products in the dual-port RAM must be prefetched. If two partial products that contribute to the same term are allowed to proceed, the second product will prefetch a sum that does not include the first product. The result is that the final sum of products will not include the first conflicting partial product.

In simpler designs, in which there is no data "Shuffler" in the PE, the sparse-BLAS controller must detect these hazards and correct them. In this case, when a data hazard is detected, the controller issues a stall command to the PE (by deasserting the *Valid* signal, and holding the values of A_{ij}, X_{jk}, and i). This behavior is shown in Figure 3.3.

3.3.2.2 Data Flow

To provide a scalable, high speed data interface between main memory and each PE, simple FIFOs are used. Each FIFO has a push/pop data interface with full and empty data signals, and is encapsulate with a ready/valid interface. Due to the sequential nature of the memory accesses, outline in Section 3.2.2, can continuously stream data into each FIFO as long as it isn't full. Additionally, by using an asymmetric FIFO (a FIFO when the read-in bitwidth in different than the read-out bitwidth), we can store an entire burst of DDR memory accesses in one clock cycle into the FIFO. This allows us to have the energy benefits of a sequential memory access with the freedom to have essentially random memory accesses between different PEs.

CHAPTER 4

FPGA Implementation

The first section covers the basics of reconfigurable computing blocks and prior attempts to use them for sparse-BLAS are also discussed. The remainder of this chapter details the development and design of two versions of the proposed architecture on a field programmable gate array (FPGA) system. The first version includes only partial support for sparse-BLAS [12]. The second version expands upon the first to provide full sparse-BLAS compatibility, with the addition of several related scalar functions, capable of executing the sparse bioinformatics algorithms presented in Chapter 2.

4.1 Field Programmable Gate Arrays

Recently, FPGAs have become an attractive option for exploring new architectures for performing sparse linear algebra [14,15,16,25,57,58,59]. FPGAs can perform trillions of floating-point operations (FLOP) per second, have large amounts of on-chip memory, and an abundant number of high-speed I/O pins capable of providing large amounts of off-chip memory bandwidth [60]. Additionally, FPGAs have demonstrated a 1 to 2 orders of magnitude improvement in energy efficiency over CPU and GPU systems [61, 62].

FPGAs are semiconductor devices whose functionality can be reconfigured, or programmed, after manufacturing. As such, they provide numerous design advantages over the fixed functionality present in standard cell Application-Specific Integrated Circuits (ASICs). FPGAs are ideally suited for the rapid prototyping and verification of VLSI systems. While ASICs take months and hundreds of thousands of dollars to design, fabricate, and verify, FPGAs can be reprogrammed in the field in a matter of seconds at a fraction of the cost [63, 64].

This added flexibility, however, does not come without a price. FPGAs incur significant penalties in area, speed, and power consumption, requiring roughly 17 to 54 times more area, run 2.5 to 6.7 times slower, and consume 5.7 to 62 times more power than a typical ASIC [65]. As detailed in Figure 4.1, this overhead is primarily due to programmable interconnects, which account for over 75% of area and delay, and 60% of the total power, in FPGAs.

To close the performance gap between FPGAs and ASICs, various circuit architectures

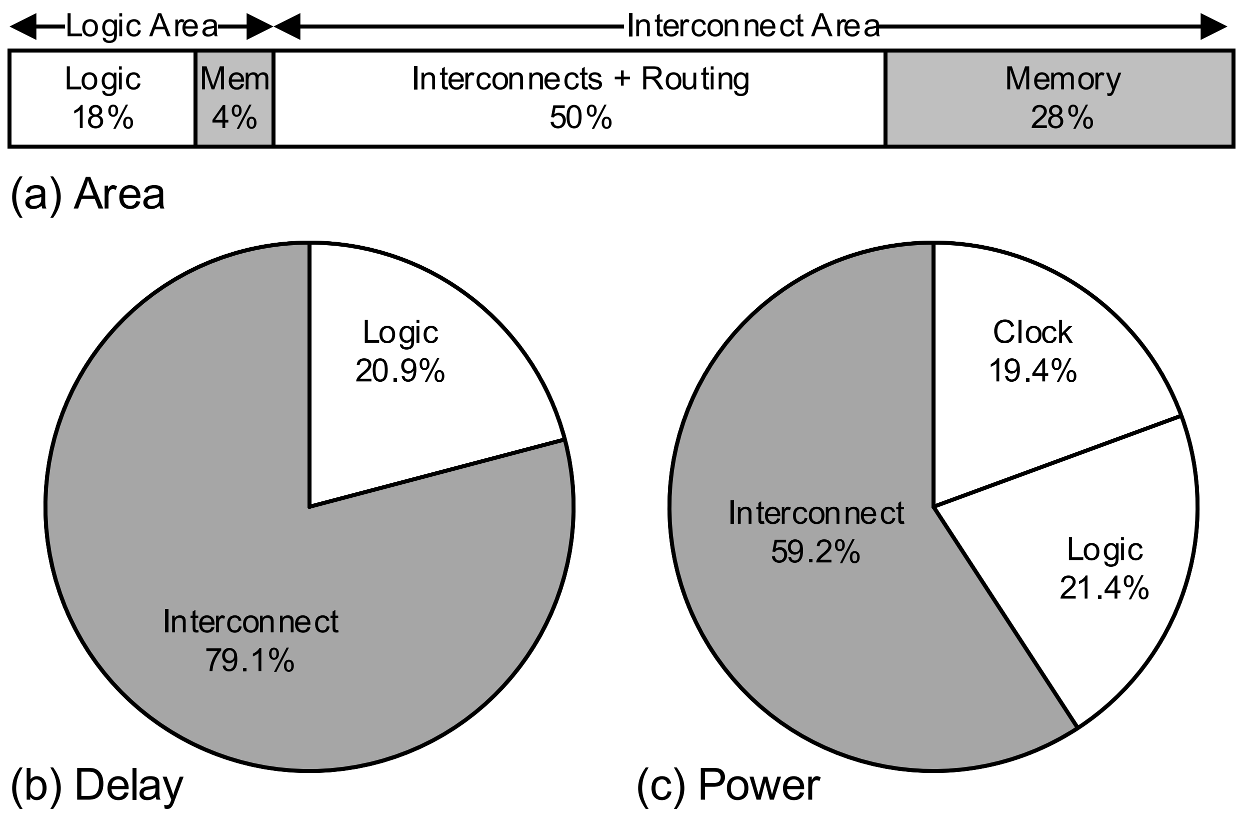


Figure 4.1: An average (a) Area, (b) Delay, and (c) Power breakdown for a simulated 90nm FPGA.

and techniques have been proposed [66, 67, 68, 69, 70].

4.1.1 Reconfigurable Computing Building Blocks

Conceptually, all FPGAs are constructed from three simple blocks. The first is a programmable or configurable logic block (CLB), which is used to implement logic functions. Second, programmable I/O blocks are needed to make off-chip connections. Finally, a programmable interconnect is required to route signals between logic blocks and I/O blocks.

Two main architectures exist for the global routing of programmable interconnects in FPGAs [63]. The first, and most widely used architecture in commercial FPGAs, is called island-style routing or a 2-D Mesh. As shown in Figure 4.2, this architecture receives its name from the way CLBs are arranged in a regular fashion on a two-dimensional grid with routing resources evenly distributed throughout it to form a mesh. The second style of

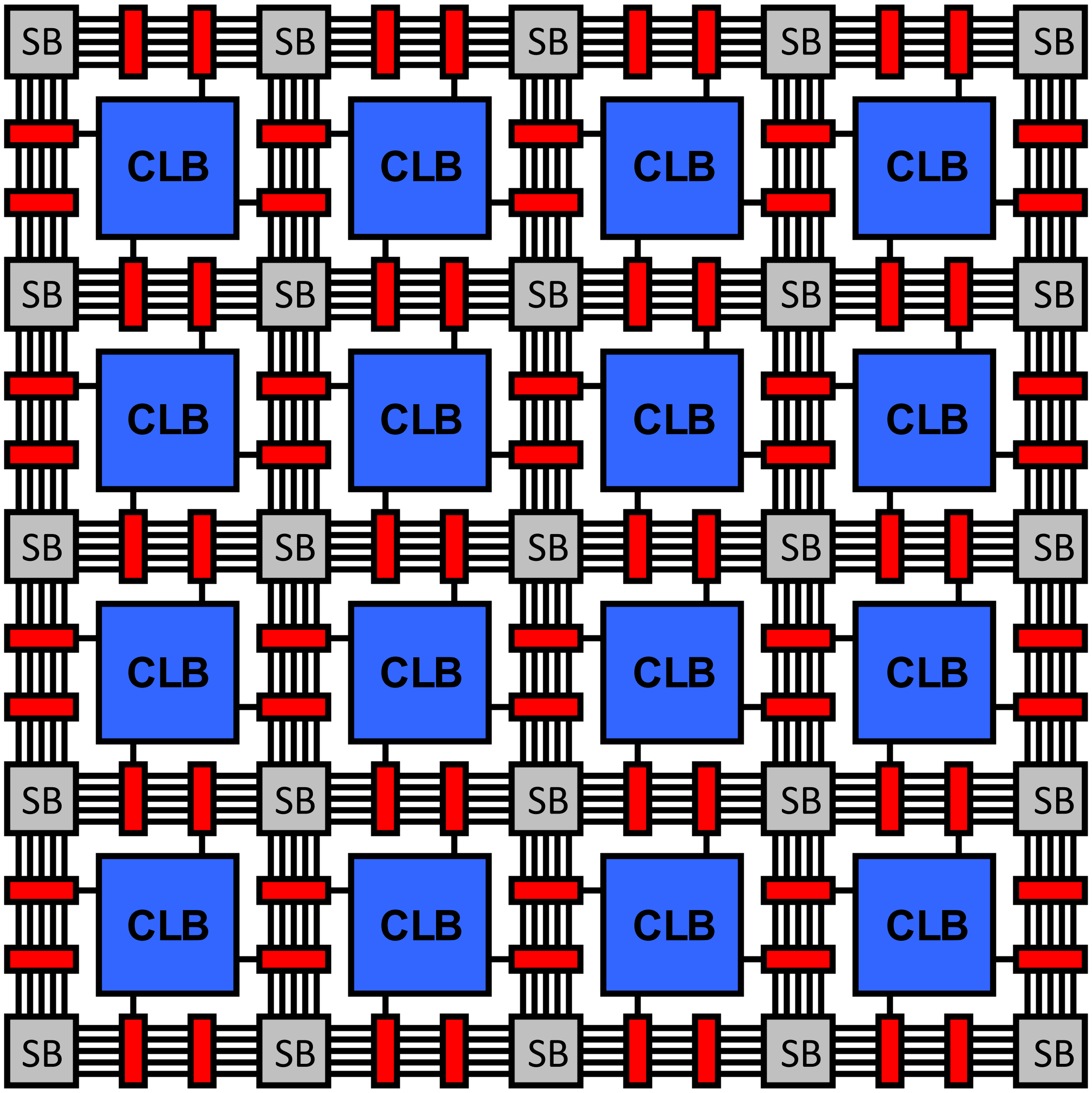


Figure 4.2: An example of an Island-Style, or 2-D Mesh, FPGA.

routing is called hierarchical routing for the way in which CLBs are organized into distinct groups or levels. As shown in Figure 4.3, only the lower most levels of routing connect to the CLB. In subsequent routing levels, the number of wires in the routing channel grows exponentially.

A CLB is usually constructed out of several logic cells: the combination of a lookup table (LUT) and a storage element. Depending upon the vender, they sometimes also called

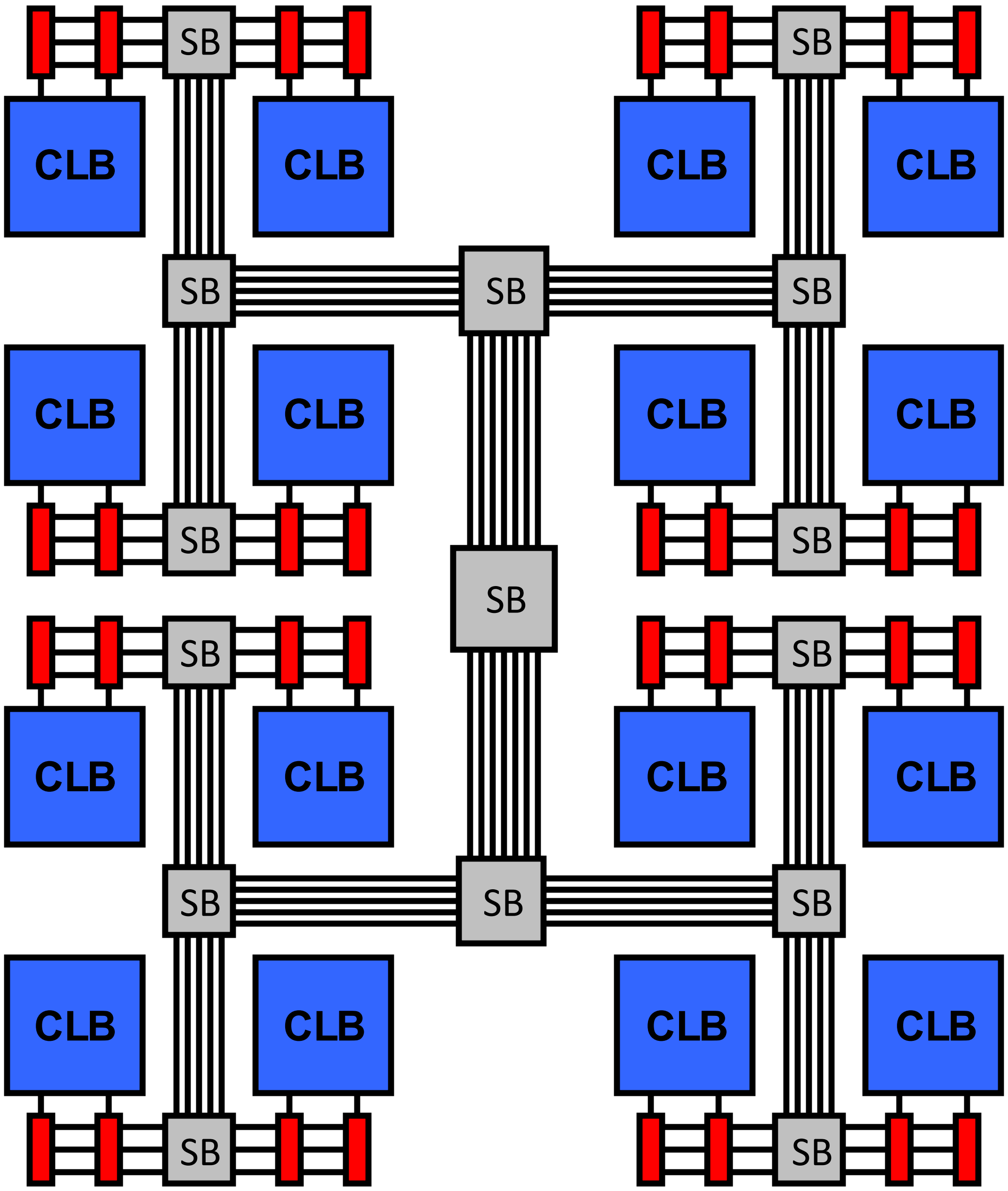


Figure 4.3: An example of a Hierarchical FPGA.

a Logic Element (LE), a Slice, or an Adaptive Logic Module (ALM). Figure 4.4 shows an example of typical logic cell employing a 4-input LUT, a full adder, and a D-type flip-flop.

The routing is accomplished by programming the various switch boxes (SBs) and switch matrices (SMs) distributed throughout the FPGA. As shown in Figure 4.5, each SB contains

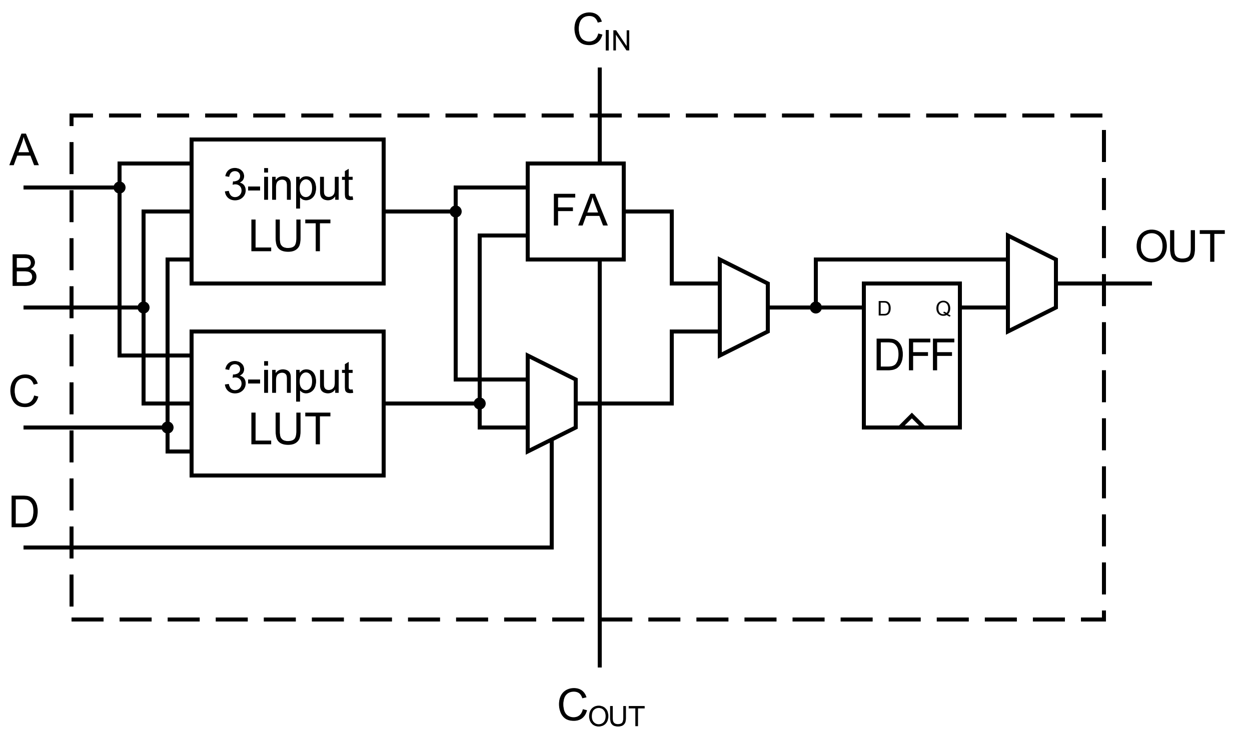


Figure 4.4: A simplified example of a logic cell with two 3-input LUT that can be configured into a single 4-input LUT, a full adder, and a D-type flip-flop.

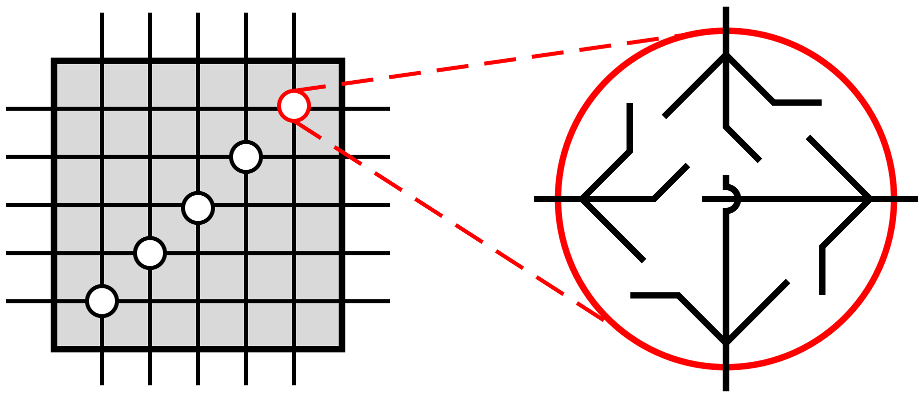


Figure 4.5: An example of a 5×5 disjoint switch block.

a set of switches to form connections between the wires segments at every horizontal and vertical intersection of the routing channels. Similarly each CLB also contains a set of programmable switched, the SM, to connect the CLB to the surrounding interconnect.

Table 4.1: Prior Art Using FPGAs

	[15]	**[14]**	**[71]**	**[58]**	**[72]**	**[12]**
FPGA	Virtex-5 LX155T	Stratix-III EP3SE260	Virtex-II Pro 70	Virtex-II Pro 100	Virtex-II 6000	Virtex-5 SX95T
Frequency [MHz]	100	100	200	170	95	150
Mem. Bandwidth [GB/s]	6.5	8.5	8	8.5	1.6	35.74
Number of PE	16	6	4	5	3	64
Peak Perf. [GFLOP/s]	3.2	1.2	1.6	1.7	0.57	19.2
Matrix Format	CVBV†	COO	CSR	CSR	SPAR*	CSC
Matrix Density						
MIN-MAX [%]	0.01-5.48	0.51-11.49	0.04-4.17	0.04-0.39	0.01-1.10	0.003-0.33
Average [%]	1.41	3.34	0.87	0.16	—	0.09
Comp. Efficiency						
MIN-MAX [%]	1-7	5-7	20-79	50-98.4	1-74	69-99.8
Average [%]	4.48	5.63	42.6	79.4	55.6	91.9

†Variant of CSR. *Variant of CSC.

4.1.2 Prior Art Using FPGAs for Sparse-BLAS

FPGA implementations have attempted to alleviate these inefficiencies by introducing several architectural changes. In some designs, several processing elements (PEs) work together to compute a single element of y in parallel [14, 25, 58]. These designs employ novel reduction circuits in order to combine the intermediary results. Other designs have each PE calculate several elements of y in a sequential manner in order to mitigate the effects of short rows [15, 57, 58]. In both cases, the entirety of the x vector, or a large subsection (in the case of blocking), is buffered in on-chip Block RAM (BRAM) to reduce the effects of irregular memory accesses [14, 15, 16, 25, 57, 58, 59]. However, these changes mainly focus on reducing the total resource usage in the design. As such, they only average around 50% of their theoretical peak processing throughput and memory bandwidth [14, 15, 16, 25, 57, 58, 59].

In Table 4.1, we compare our proposed SpMxV architecture to several published SpMxV FPGA architectures. The architectures can be categorized into 3 distinct approaches. The first is to either re-encode, reorder, or preprocess the sparse matrix in such a way as to reduce data hazards [14, 15] (see Figure 4.6). Kestur et al. [15] re-encode the matrix into the Compressed Variable-Length Bit Vector (CVBV) format (a variation of the CSR format) on the fly to reduce the required memory bandwidth. However, re-encoding the matrix insures a significant overhead penalty, resulting in marginal efficiency improvements over CPU and

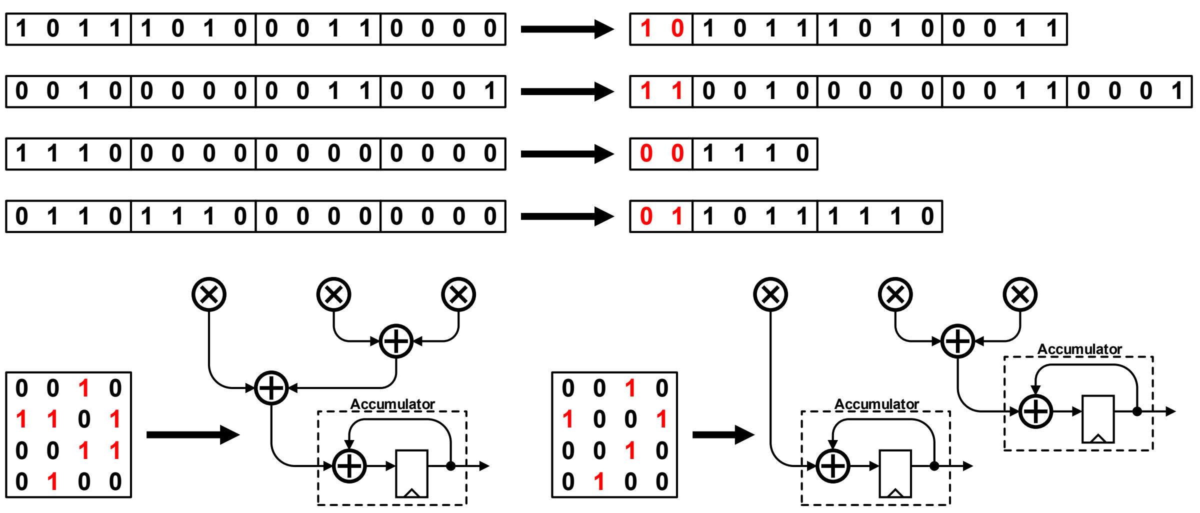


Figure 4.6: Examples demonstrating preprocessing on the sparse matrix re-encode or re-order the matrix in such a way as to reduce either the data rate of the number of data hazards.

GPU implementations (only 3-10x). Sun et al. [14] preprocess the matrix to reorganize and optimize the datapath to eliminate hazards. After preprocessing, the matrices achieve a computational efficiency of 96-99% on the FPGA. Unfortunately, the preprocessing overhead (datapath optimization, FPGA reconfiguration, and buffering the matrix on BRAM) is about 20 times greater than that of the SpMxV calculation. This results in an effective computational efficiency of only 5-7%.

The second approach is to use several PE (each with its own working copy of x) in parallel with a novel reduction circuit or adder tree to accumulate a single element of y [58, 71]. In Zhuo et al. [71], the partial product of 4 multipliers are added together via a tree of 3 adders. The resulting sum is then fed into a novel reduction circuit (accumulator) that handles the potential read-after-write data hazards. The drawback of this approach is that requires zero padding to achieve a minimum row size. Combined with blocking along the columns of A, the design is sensitive to the sparsity structure of the matrix. The computational efficiency ranges from 20% to 79%, performing particularly poorly for extremely sparse matrices (<0.1% density). Zhang et al. [58] improves upon this design by having each PE calculate a different element of y. Their accumulator requires a minimum row length of 8, zero padding when necessary, but can switch between any of several different rows assigned to it if it encounters a data hazard. This roughly doubles the computational efficiency of the design as compared to Zhuo et al. [71]. However, it still performs quite poorly for extremely sparse matrices.

The final approach is to use the method outlined in Chapter 3 to stream CSC matrix data through several PE. Gregg et al [72] uses a variant of CSC called the sparse matrix architecture and representation (SPAR) format. In the SPAR format, *row* and *ptr* are combined into a single vector with zero padding introduced at the start of each column of the data vector. Each PE only buffers a small portion of the y vector (32 elements vs. 3,456 elements in our design) in a local cache. A y-cache miss requires a write-back to high latency DRAM, incurring a 109 clock cycle penalty. With such a small cache, the design is very highly sensitive to the sparsity structure of the matrix. Computational efficiency ranges from 1% to as high as 74%.

4.1.3 Reconfigurable Open Architecture Computing Hardware (ROACH)

The SpMxV and sparse-BLAS kernels were prototyped and demonstrated on the open-source academic research platform "ROACH" (Reconfigurable Open Architecture Computing Hardware) [73]. The "ROACH" was developed as part of an international collaboration of primarily radio astronomy research institutions as a common high-performance computing platform for real-time digital signal processing applications [74]. A block diagram of the system is shown in Figure 4.7. A picture of the "ROACH" FPGA system is shown in Figure 4.8.

This ROACH platform uses a Virtex-5 SX95T FPGA that has been component optimized for DSP applications. This particular FPGA is equipped with a large number of embedded multipliers ("DSP48E") and dual-ported SRAM blocks ("BlockRAMs" or "BRAMs"). In addition to its 14,720 logic SLICEs[1], an SX95T has 640 DSP48Es, and 488 18-kbit BRAMs [75]. A Virtex-5 DSP48E is a "hard macro": that is, it is a circuit-level implementation of a 25×18-bit multiplier-accumulator, and consumes no reconfigurable logic resources in order

[1]A Virtex-5 SLICE is four 6-input lookup tables (LUTs) and four flip-fops (FFs).

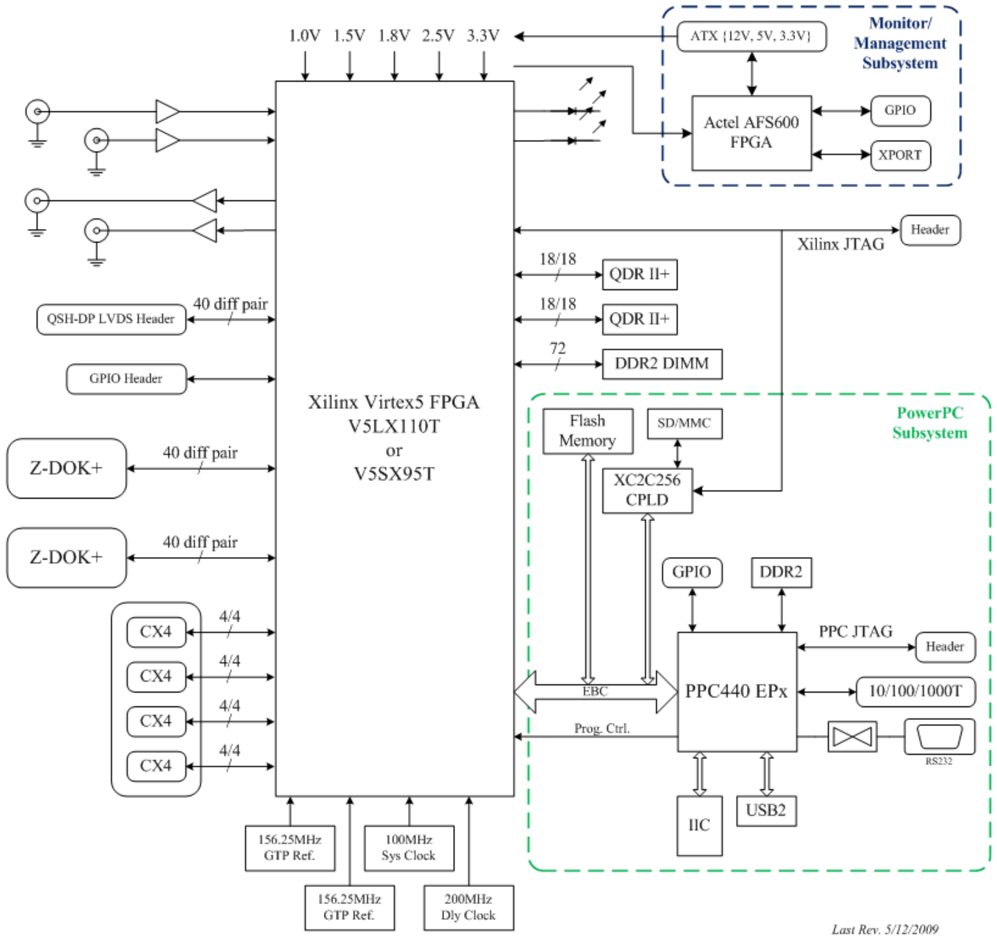


Figure 4.7: A block diagram of the open-source academic research platform "ROACH" (Reconfigurable Open Architecture Computing Hardware) [73].

to create a multiplier, adder, or multiply-accumulate unit (MAC).

The "ROACH" system also has two 36Mb QDRII+ SRAMs and 2GB of DDR2 SDRAM for a combined peak memory bandwidth of 35.74 GB/s. The "ROACH" uses ECC (error-correcting code) DDR2 memory modules, with each module presenting a 72-bit interface due to an extra parity byte. The FPGA's DDR2 controller provides a 144-bit, two-word-burst interface to the FPGA fabric. Due to the way that the way DDR2 memory is implemented at the physical level, the "ROACH" can only support operating the modules at a limited number of frequencies: 150, 200, 266, 300, or 333 MHz.

The QDR memories on "ROACH" have 18-bit data buses with a 4-word DDR burst of consecutive memory locations. As with the DRAM used, the non-multiple-of-eight data

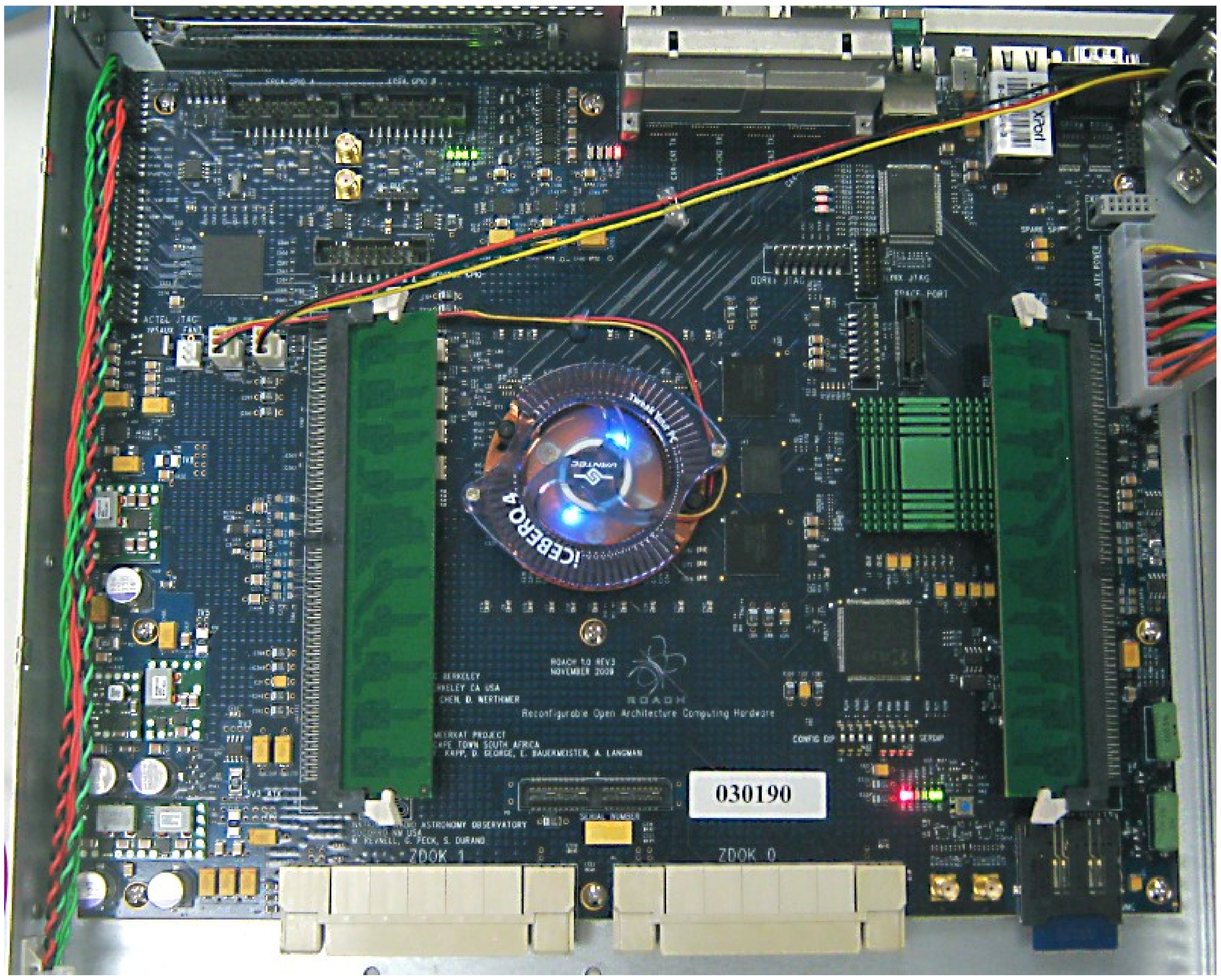


Figure 4.8: A picture of the open-source academic research platform "ROACH" (Reconfigurable Open Architecture Computing Hardware) [73].

width is nominally for ECC parity bits; in both interfaces, 9 bits is considered to be a single byte. The FPGA's QDR controller abstracts this physical layer and presents an interface with a two-word burst of 36 bits each. The SRAM structure allows the controller to have a fixed 10-cycle read latency and 1-cycle write latency. The QDR chips have the flexibility to run at any clock frequency between 150 and 400 MHz.

The "ROACH" system is controlled by an on-board PowerPC 440EPx embedded processor with 512 MB of dedicated DDR2 DRAM, non-volatile Flash memory, and a Gigabit Ethernet interface. This allows the PowerPC to run a full Linux operating system and enables a computer running MATLAB to remotely interface with "software registers," BRAMs,

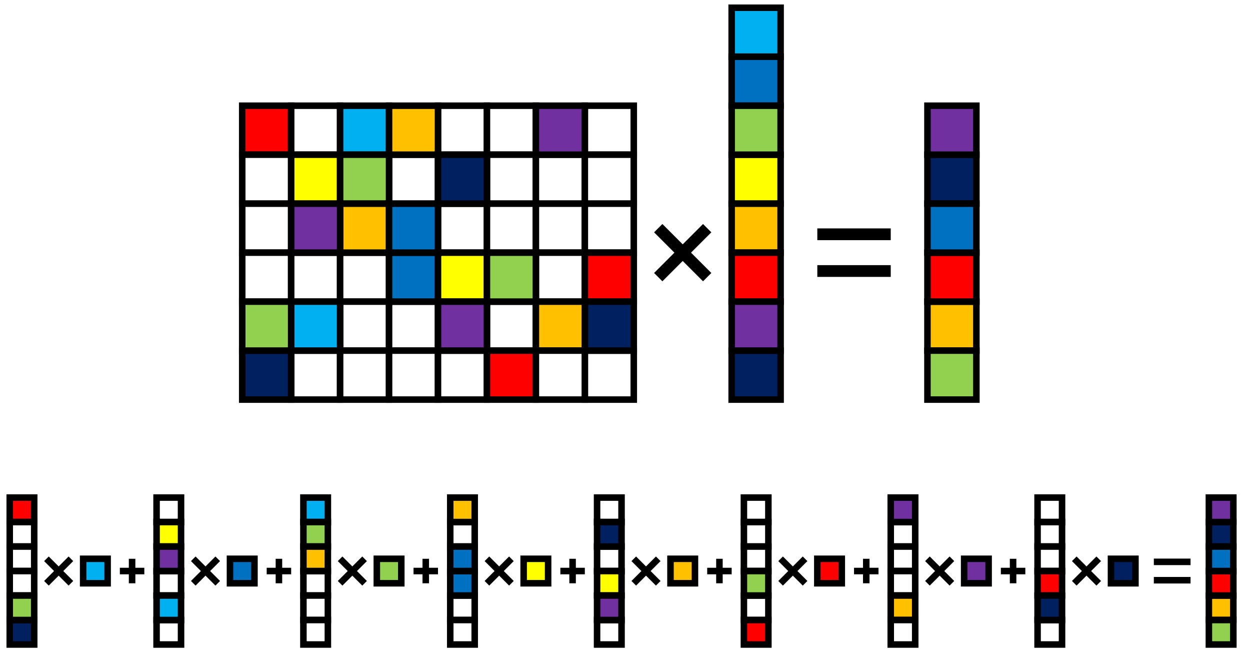


Figure 4.9: A graphical representation of how SpMxV is performed using the CSC format. The entire vector y is calculated as a series of vector additions of the columns of A weighted by the appropriate element from x.

and FIFOs on the FPGA, as well as to load data in and out of the QDRs and SDRAM on the board.

4.2 SpMxV Architecture

Our first prototype, an FPGA architecture implements only up to level 2 sparse-BLAS operations (Equations 2.1 and 2.2). Using the CSC data format, the FPGA architecture abandons the idea of calculating each element of y separately as the row-wise dot product between A and x (as shown in Figure 2.6). Instead, the entirety of y is calculated as the column-wise vector additions of A weighted by each element of x, as shown in Figure 4.9. Fundamentally, this allows us to directly address the major limitations present in current CPU and GPU systems for the SpMxV algorithm:

(1) An FPGA co-processor allows for much better balancing of system resources. The number of processing elements (PEs) and operating frequency can be tailored to match

the available memory bandwidth to optimize energy efficiency.

(2) What used to be indirect (global) memory references for x in the *col* vector (for the CSR format) are now direct (local) memory references for y in the *row* vector. In other words, when a column of A is weighted by an element of x, we know exactly which elements of y the partial products will contribute to. This allows us to halve the number of required memory accesses, one of the largest bottlenecks in the SpMxV and SpMxM kernels.

(3) Memory access to the x vector is no longer irregular, but sequential. By using the CSC format to store A, both A and x can be placed in a large off-chip memory and sequentially streamed into the co-processor (eliminating the time and energy overheads of a cache miss present in CPUs and GPUs).

(4) Short row or column lengths have much less impact on the performance, since the PEs are rarely idled thanks to the balanced memory bandwidth and computing capability. However, the performance can be degraded if the memory bandwidth of x approaches that of A, as in the case of extremely sparse matrices (nonzero densities <0.001%). In the rare case of $M \gg N$, the performance of CSC is no better than that of CSR due to the significant increase in the memory bandwidth required by x.

Figure 4.10 shows the overall architecture of the SpMxV kernel, which serves as a co-processor attached to an external computer. A dedicated memory controller allows the elements of A and x to be continuously streamed into the FPGA, while the SpMxV controller's primary function is scheduling and hazard detection. Hazard detection avoids the conflict between two partial products that contribute to the same element of y overlapping due to the latency of the floating-point adder. Each of the 64 PEs contain a single-precision floating-point adder and multiplier, as well as a simple dual-port RAM for partial product accumulation.

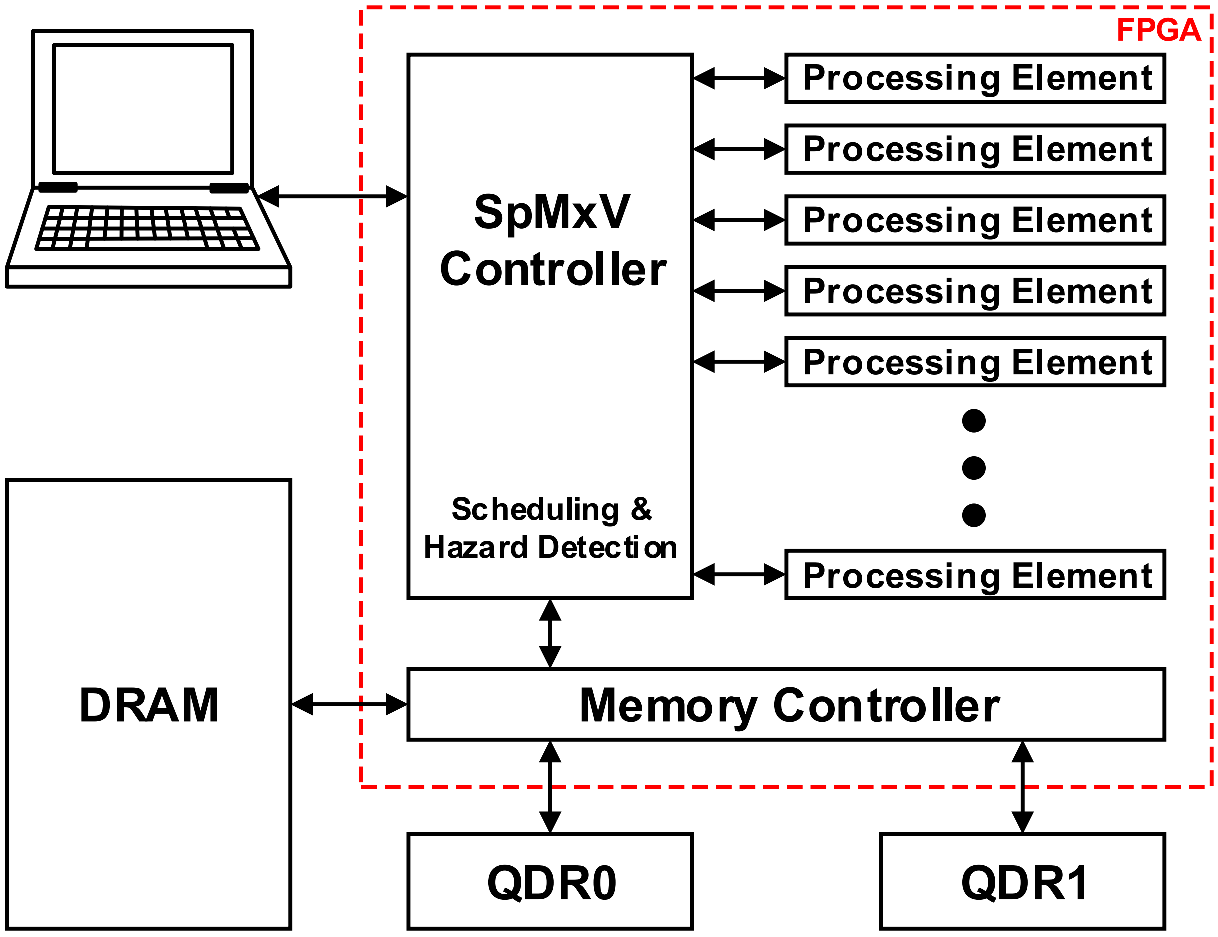


Figure 4.10: Top-level schematic of the SpMxV kernel, containing 64 single-precision floating-point processing elements, running on the "ROACH" FPGA platform [73]

4.2.1 Processing Element

As stated previously, each PE (Figure 4.11) contains a single-precision floating-point adder and multiplier, as well as a simple dual-port RAM. To perform the SpMxV, each PE multiplies an element of the data vector (A_{ij}) and the corresponding element of the x vector (X_j) together. The resulting partial product is then added to address in the row vector (i), before being stored back into the BRAM of the dual-port RAM. Due to the latency of the multiplication and addition operations, a *Valid* signal is used to prevent data corruption due to hazards. Using this strategy, data can be continuously streamed into each PE (directly from the CSC format) with a small startup overhead latency equal to that of the adder and

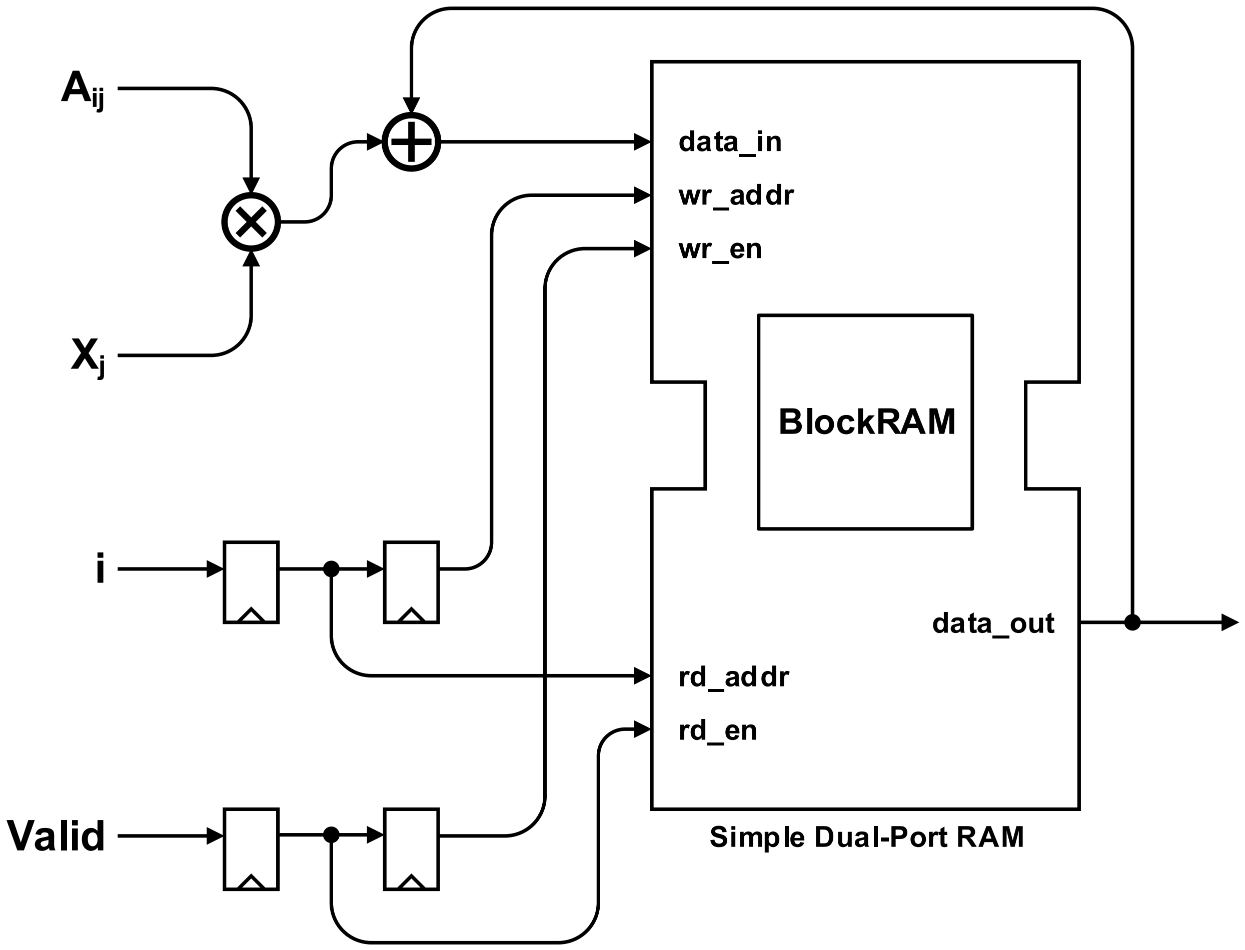


Figure 4.11: Block diagram of the proposed processing element without data reordering.

the multiplier.

Since each PE has its own working copy of the vector being computed, there are two possible strategies for assembling the final vector. The first option is to assign a subset of the x vector to each PE (i.e. blocking along the columns of A). Each PE computes a partial sum of the final vector and an adder tree (which can be built from the existing adders in each PE) is used to combine them at the end. Similar to prior FPGA implementations [14, 15, 16, 25, 58, 71, 72], this straightforward approach has several drawbacks. First, the reduction circuit adds a large amount of overhead in terms of latency and additional hardware (even if existing adders are used, more resources are needed for reconfigurability). Second, by splitting up computation along the columns, we lose some of the sequential nature of the memory accesses we had gained with the CSC format. The memory accesses for each PE are still sequential, but globally the memory accesses for all PEs are irregular. To mitigate this, a more complicated memory controller is required to ensure a balanced load across all of the PEs. Third, if there are fewer columns than PEs, this approach is effectively no different than prior FPGA implementations.

The second option for computing the final vector is to assign a subset of rows of A to each PE (i.e. blocking along the rows of A). Each PE computes a subset of the final vector, which are then concatenated together at the end (requiring no additional latency). Additionally, this preserves the property of sequential memory accesses across all PEs, allowing for a much simpler memory controller (at the cost of a slightly more complicated SpMxV controller to handle additional scheduling and hazard detection). Finally, with minor modifications to the SpMxV and memory controllers, this SpMxV kernel can be made to also support SpMxM: each column of the dense matrix can be assigned to a PE, which each PE computing a single column of the resulting matrix. This is implemented in our second sparse-BLAS, FPGA-based architecture.

4.2.2 Data Hazard and Memory Management

The primary purposes of the sparse-BLAS controller are scheduling and hazard detection. The memory controller acts as a slave to the sparse-BLAS controller, ensuring a continuous stream of data into the PEs. Hazards arise when two or more partial products want to write to the same memory address of the dual-port RAM in a short period of time. Due to the latency of the floating-point adder of the PE, shown in Figure 4.11, the existing sum of the partial products in the dual-port RAM must be prefetched. If two partial products that contribute to the same term are allowed to proceed, the second product will prefetch a sum that does not include the first product. The result is that the final sum of products will not include the first conflicting partial product. The sparse-BLAS controller detects these hazards and corrects them by issuing a stall command to the PE (by deasserting the *Valid* signal, and holding the values of A_{ij}, X_jk, and i). Figure 3.3 shows this kind of stalling behavior for a single processing element performing SpMxV between the example A matrix

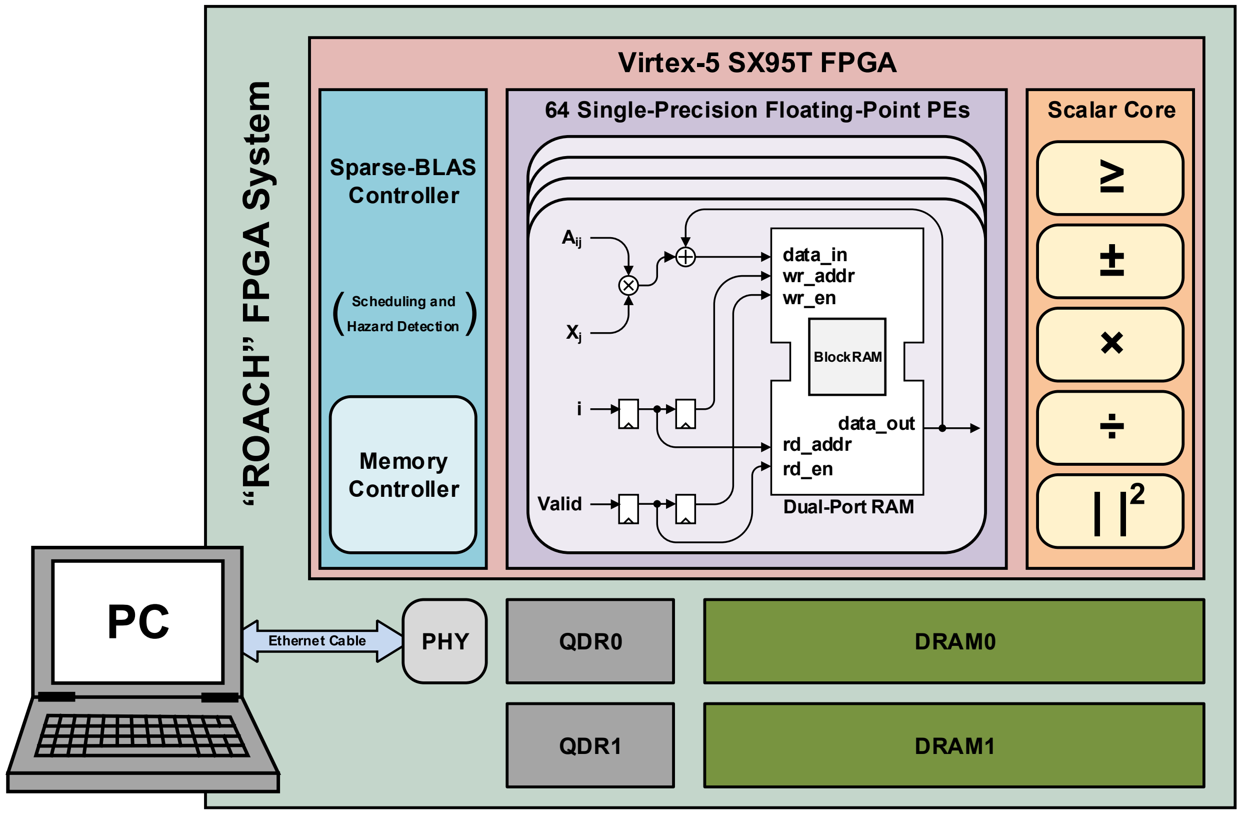


Figure 4.12: Top-level schematic of the sparse-BLAS kernel, with a vector core containing 64 single-precision floating-point processing elements and a scalar core for vector-vector and vector-scalar operations, running on the "ROACH" FPGA platform [73]. The "ROACH" FPGA system runs a bare-bones Linux kernel and acts as a co-processor for a networked computer running a MATLAB environment.

from Figure 2.1 and the vector $x = [0.1 \;\; 0.2 \;\; 0.3 \;\; 0.4 \;\; 0.5]^T$.

4.3 Sparse-BLAS Architecture

Figure 4.12 shows the overall architecture of the sparse linear algebra kernel, which serves as a co-processor attached to an external computer. Computing SpMxV and SpMxM column-wise allows us to retain an extremely simple PE. In CPUs and GPUs, the main computational element is a Multiply-Accumulate Unit (MAC). Each MAC consists of a floating-point adder and multiplier with an accumulation register. In our architecture, the each PE in the vector core (Figure 4.12) contains a single-precision floating-point adder and multiplier, as well as a simple dual-port RAM. The simple dual-port RAM replaces the accumulation register in the MAC and can accommodate several hundred to several thousand elements of y, eliminating the need for costly off-chip memory accesses.

Additionally, data in the dual-port RAM can be accessed in a single clock cycle, compared to roughly 30 clock cycles for a L3 cache in a CPU and more than 200 clock cycles for a L3 cache in a GPU. And finally, since the memory accesses are sequential with no data dependences, our architecture has an effective FLOP to memory fetch ratio of 1-to-0, giving a theoretical computational efficiency of 100%.

A dedicated memory controller allows the elements of A and x to be continuously streamed into the FPGA, while the sparse-BLAS controller's primary function is scheduling and hazard detection. Hazard detection avoids the conflict between two partial products that contribute to the same element of y overlapping due to the latency of the floating-point adder. If such a hazard is detected, we must either stall or provide alternative data to ensure that the result of y is calculated correctly. A small scalar core is included to provide support for vector-vector and vector-scalar operations like addition, subtraction, element-wise division and multiplication, ℓ_1 and ℓ_2 norms, absolute value, Hadamard power, and logical comparisons.

4.3.1 Processing Element

As stated previously, each PE contains a single-precision floating-point adder and multiplier, as well as a simple dual-port RAM (see Figure 4.11). To perform SpMxV, each PE multiplies an element of the data vector (A_{ij}) and the corresponding element of the x vector (x_j) together. The resulting partial product is then added to the address in the row vector (i), before being stored back into the block-RAM (BRAM) of the dual-port RAM. Due to the latency of the multiplication and addition operations, a *Valid* signal is used to prevent data corruption due to hazards. Using this strategy, data can be continuously streamed into each PE (directly from the CSC format) with a small startup overhead equal to the combined latency of the adder and the multiplier.

Since each PE can only store a finite number of elements (several hundred to several thousand depending upon the FPGA), each PE contains a partial working copy of the vector being computed. To facilitate SpMxV, blocking is performed along the rows of A (i.e. each PE is assigned a subset of rows of A for computation). The final vector is then assembled by concatenating the output of each PE during memory write-back (requiring no additional latency). This parallelization strategy (essentially a block CSC matrix storage format) preserves the property of sequential memory accesses across all PEs to ensure high computational efficiency. It also allows for the same hardware to perform SpMxM: a different column of the X matrix (Equation 2.3) is assigned to each PE, with each PE computing a single column of the resulting matrix using the same A_{ij} but different X_jk data (where k is the column assigned to each PE).

4.3.2 Data Hazard and Memory Management

The primary purposes of the sparse-BLAS controller are scheduling and hazard detection. The memory controller acts as a slave to the sparse-BLAS controller, ensuring a continuous stream of data into the PEs. Hazards arise when two or more partial products want to write to the same memory address of the dual-port RAM in a short period of time. Due to the latency of the floating-point adder of the PE, shown in Figure 3.7, the existing sum of the partial products in the dual-port RAM must be prefetched. If two partial products that contribute to the same term are allowed to proceed, the second product will prefetch a sum that does not include the first product. The result is that the final sum of products will not include the first conflicting partial product. The sparse-BLAS controller detects these hazards and corrects them by issuing a stall command to the PE (by deasserting the *Valid* signal, and holding the values of A_{ij}, X_{jk}, and i). Figure 3.3 shows this kind of stalling behavior for a single processing element performing SpMxV between the example A matrix from Figure 2.1 and the vector $x = [0.1\ \ 0.2\ \ 0.3\ \ 0.4\ \ 0.5]^T$.

CHAPTER 5

ASIC Implementation

This chapter discusses the details of the ASIC implementation of sparse-BLAS architecture. Details of the processing element, including the data "shuffler," and the memory control scheme for achieving 100% computational efficiency (i.e. utilization) are explained. Design considerations for chip testing and configuration are discussed in the final section.

5.1 Overall Architecture

Figure 5.1 shows the top level schematic of the ASIC sparse-BLAS kernel. It consists of a sparse-BLAS controller, with an integrated memory controller, four dedicated processing elements for the CSC data format, a configuration scan chain, and a 512-kbit dual-port SRAM for testing. The following sections with look into the details of each of these blocks.

5.2 Processing Element

Each PE contains a single Floating-Point Unit, as well as a simple dual-port RAM (implemented as a 512-word, 32-bit dual-port SRAM) and a data-stream reorderer (or "Shuffler"). Figure 5.2 shows a block diagram of the PE. Figure 5.3 shows a layout view of the fabricated PE. To perform sparse-BLAS, each PE multiplies an element of the data vector (A_{ij}) and the corresponding element of the x vector (x_j) together. The resulting partial product is then added to the address in the row vector (i), before being stored back into the dual-port RAM. Due to the latency of the multiplication and addition operations, a *Valid* signal is used to prevent data corruption due to hazards. The "Shuffler" attempts to reduce the number of data hazards by first monitoring for them, and then swapping in alternative partial products (instead of stalling) until the data hazard is resolved. Using this strategy, data can be continuously streamed into each PE (directly from the CSC format) with a small startup overhead equal to the combined latency of the adder, multiplier, and "Shuffler".

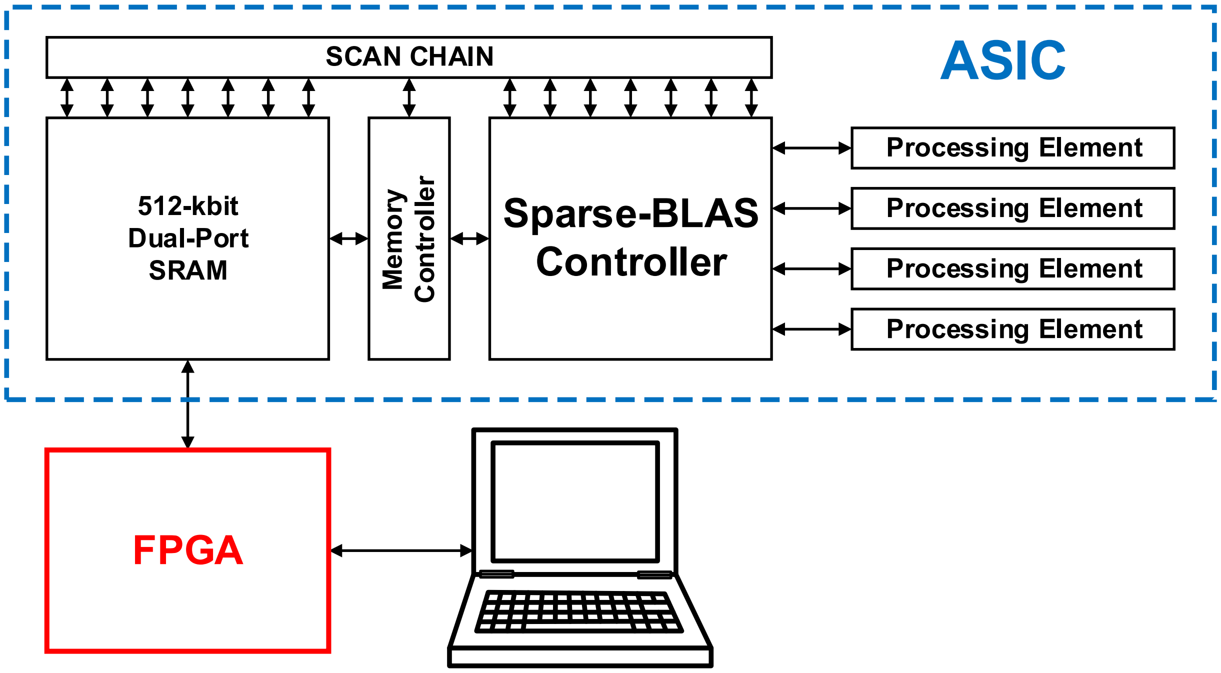


Figure 5.1: Top-level schematic of the sparse-BLAS ASIC chip.

5.2.1 Floating-Point Unit

The FPU of each PE contains a floating-point adder and multiplier. The purpose of the FPU is to perform the partial product multiplication and accumulation for each element of the data vector (A_{ij}) and the corresponding element of the x vector (x_j). In this work, we use the single-precision floating point format (binary32) specified by the IEEE 754-2008 standard [56]. We use the default rounding mode: round to nearest, ties to even.

5.2.2 Dual-Port Memory

Since each PE can only store a finite number of elements, each PE will need to contain a partial working copy of the vector being computed. To facilitate this, blocking is performed along the rows of A (i.e. each PE is assigned a subset of rows of A for computation). The final vector is then assembled by concatenating the output of each PE during memory write-back (requiring no additional latency). This parallelization strategy (essentially a block CSC matrix storage format) preserves the property of sequential memory accesses across all PEs

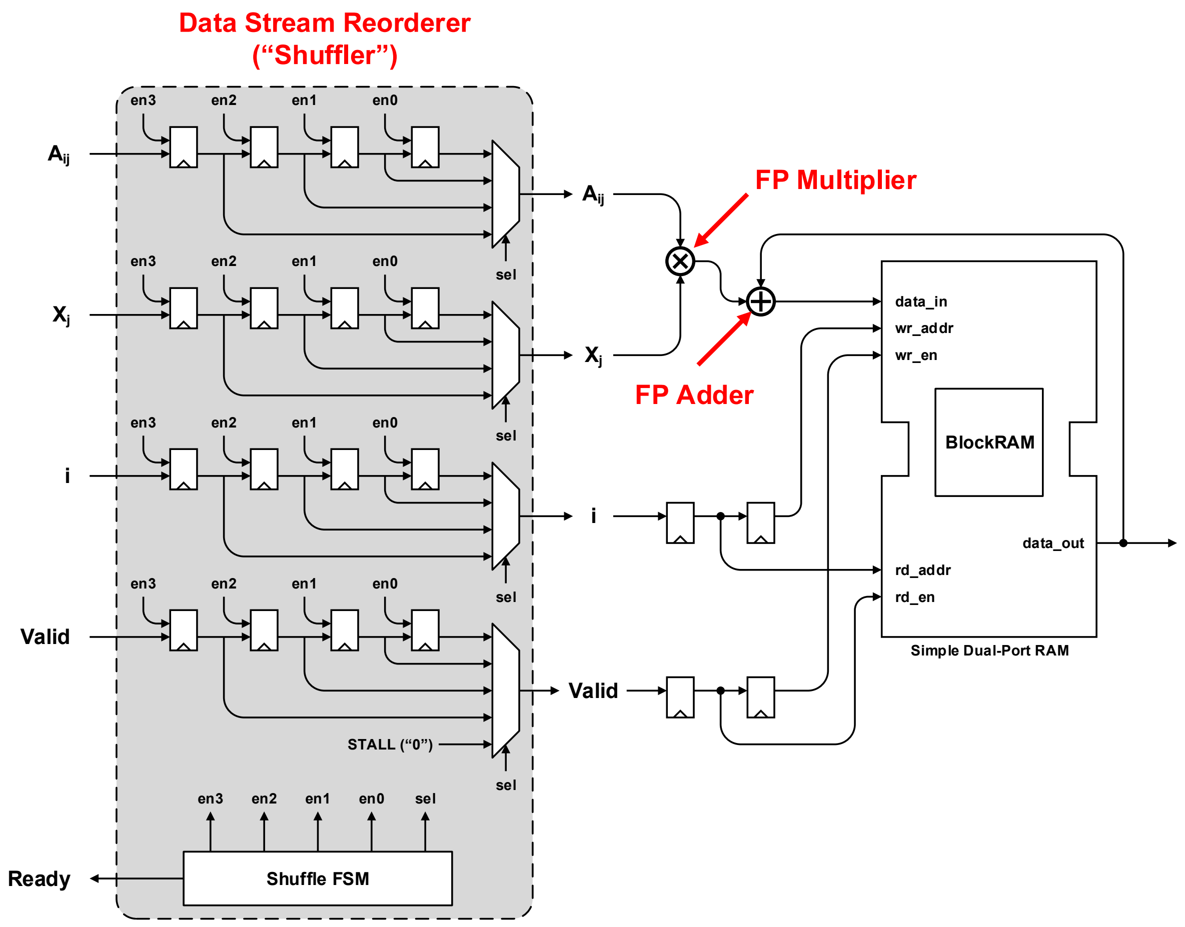


Figure 5.2: Block diagram of the fabricated processing element with data reordering.

to ensure high computational efficiency.

To ensure that we can both compute and update the partial products in the same clock cycle, a dual-ported memory is required. The first port of the memory is set to a read-only state to provide data for the partial product accumulation. This port is also used to write out the contents of the dual-ported memory to main system memory. The second port is fixed into a write only state, with a write enable signal, to write back the updated partial products. This port can also be used to zero-out the memory, or load a predefined vector in the most general case of SpMxV and SpMxM.

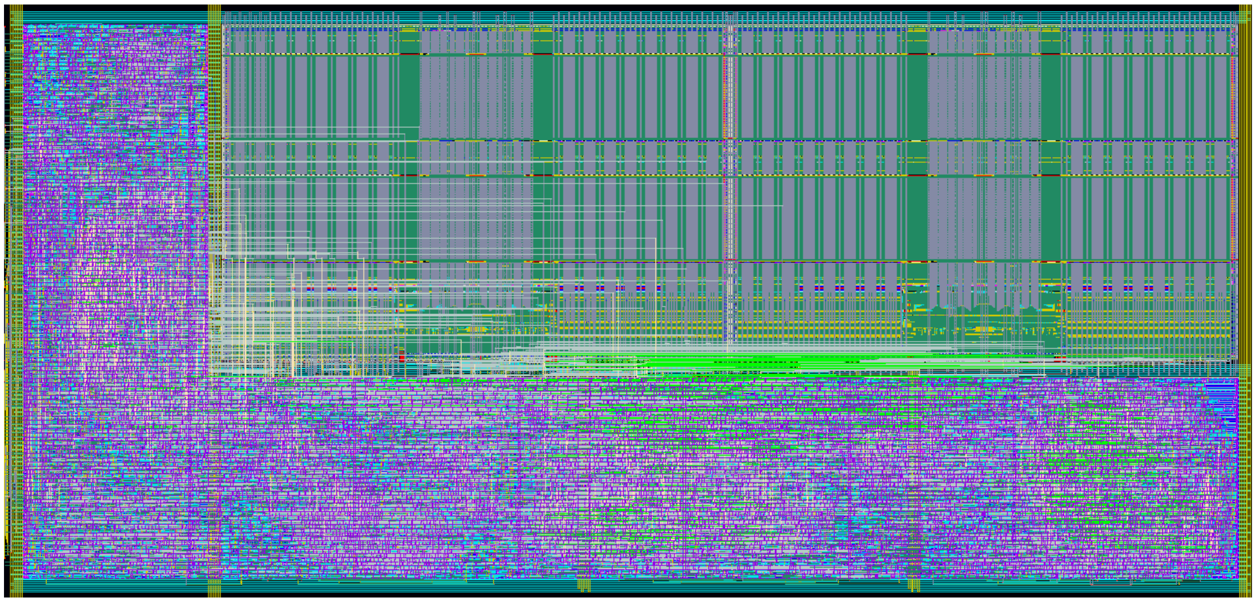


Figure 5.3: Layout view of the fabricated processing element with data reordering.

5.2.3 The “Shuffler”

The data-stream reordering unit, or “Shuffler, consists of a buffer for four items each of A_{ij}, X_j, i, and $Valid$ data used to calculate a partial product, and a FSM to control the flow of data in the buffer. The FSM monitors each piece of data in the buffer to see if it will create a data hazard further down the pipeline if it is issued to the FPU. Every clock cycle the FPU requests a new piece of data. The FSM will issue the first element in the buffer that will not create a data hazard. If the buffer is empty or every valid piece of data in the buffer would create a data hazard, the FSM will issue a stall command to the FPU. An example of this type of behavior can be seen in Figure 3.4.

5.3 Sparse-BLAS Controller

The primary purpose of the sparse-BLAS controller is scheduling and controlling data flow between each of the four PEs and the 512-kbit dual-port testing memory.

5.3.1 Memory Controller

The memory controller acts as a slave to the sparse-BLAS controller, ensuring a continuous stream of data into the PEs. To provide a scalable, high speed data interface between main memory (the 512-kbit dual-port SRAM) and each PE, simple FIFOs are used. Each FIFO has a push/pop data interface with full and empty data signals, and is encapsulate with a ready/valid interface. Due to the sequential nature of the memory accesses, outlined in Section 3.2.2, can continuously stream data into each FIFO as long as it isn't full. Additionally, by using an asymmetric FIFO (a FIFO when the read-in bitwidth in different than the read-out bitwidth), we can store an entire burst of DDR memory accesses in one clock cycle into the FIFO. This allows us to have the energy benefits of a sequential memory access with the freedom to have essentially random memory accesses between different PEs.

5.4 Testing and Configuration Considerations

This section covers a number of design considerations for the ASIC implementation in order to facilitate an automated FPGA testing platform with the ability to mimic the memory conditions expected on an SoC.

5.4.1 FPGA Interface

We use the same "ROACH" FPGA system to test the ASIC implementation [73]. Using the on-board PowerPC system running Linux, we are a able to use a computer running MATLAB to remotely interface with "software registers," BRAMs, and FIFOs on the FPGA. This also allows us to load data in and out of the QDRs and SDRAM on the board, as well as the two Z-DOK+ interfaces used to communicate with the ASIC PCB test board. A block diagram of the system is shown in Figure 4.7. A picture of the "ROACH" FPGA system is shown in Figure 4.8. A picture of the ASIC test PCB is shown in Figure 5.4.

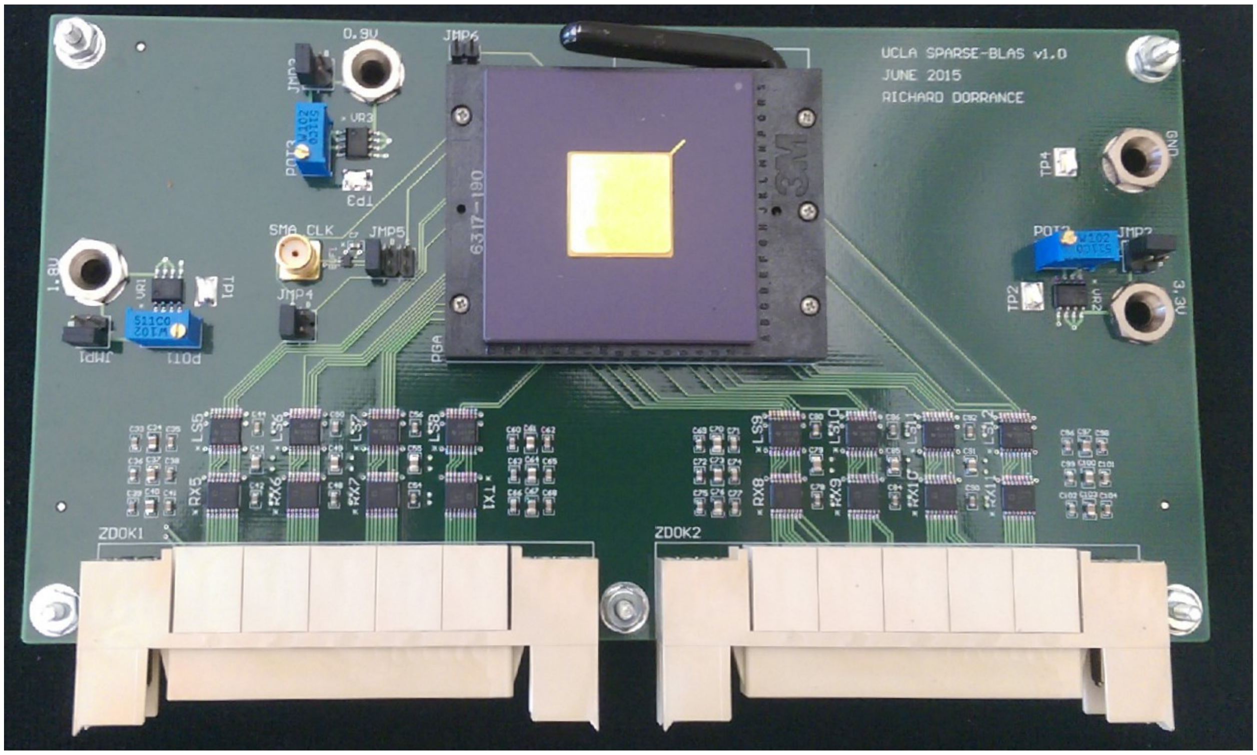


Figure 5.4: A picture of the PCB board used to test the sparse-BLAS ASIC.

5.4.2 Scan Chain

In order to reduce the number of digital I/O required for testing the ASIC implementation of the sparse-BLAS kernel, a 323-bit scan chain was used. Each processing element requires 64 bits of configuration: the number of elements to process when a command is issued and four memory addresses (corresponding to the three arrays of the CSC data format *data*, *row*, and *ptr*, and the x vector). Additionally, 3 bits are used to configure the command that is issued simultaneously to each of the PE when the *start* signal is asserted (IDLE = 000, LOAD = 001, READ = 010, MULTIPLY = 011, and ZERO OUT = 101). The last 64 bits are used as a clock cycle counter to precisely measure the number of clock cycles used to finish executing a command.

5.4.3 Memory Cache

A 512-kbit dual-port SRAM is used as a memory cache for the main RAM (either the DDR2 or QDR memories) implemented on the FPGA. The Cache memory on the ASIC chip emulates a DDR2 memory interface in order to simulate a much larger DDR2 memory that would be expected if the kernel were integrated into an SoC. The cache uses a simple sequential replacement policy for the cache's eight memory blocks, which, due to the sequential nature of the CSC data accesses, effectively results in a Least Recently Used (LRU) replacement policy.

The dual-port SRAM cache also allows us to neatly partition into two clock domains: the FPGA interface and the sparse-BLAS ASIC core. By decoupling the testing and active portions of the ASIC chip, we are able to fully characterize the power and speed of the sparse-BLAS hardware, with voltage scaling, without having to redesign and recompile the FPGA interface for every testing scenario. This allows for a highly automated testing infrastructure to reliably and repeatedly test multiple ASIC chips.

CHAPTER 6

Testing and Performance Results

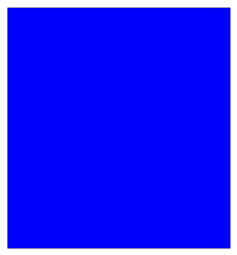
In order to measure the performance of the sparse-BLAS kernel we will be using a variety of different data sets to measure the computational and energy efficiency of the proposed architecture, as well as the raw performance. These data sets are presented in Section 6.1. Section 6.2 introduces the various software and hardware test platforms that comparisons are made to. The final Sections 6.3 and 6.4 present performance results and energy efficiency results for two FPGA implementations and our ASIC implementation.

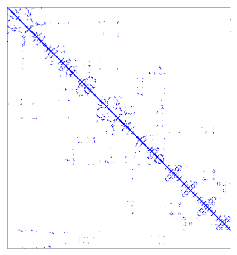
6.1 Data Sets

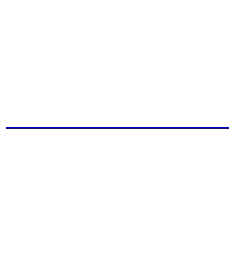
This section introduces and discusses the data sets used in our evaluation of the sparse-BLAS kernels. They consist of two major types of data sets: general sparse matrices and biological data sets. The general sparse matrices are used to benchmark the computational and energy efficiency of the sparse-BLAS hardware. The biological data sets are used to evaluate the performance of several bioinformatics algorithms.

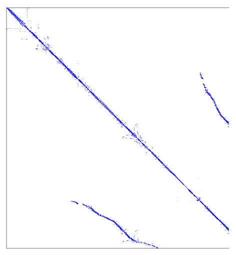
6.1.1 Sparse Matrix Collection

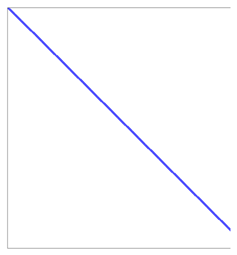
We use a collection of 14 unstructured matrices used by both Williams et al. [1] and Bell et al. [2] in our performance benchmarking study. Table 6.1 details the size and overall sparsity structure of each matrix. The matrices range from 100% density to densities as low as 0.00031%, with as many as 11.5+ million nonzero elements. The largest matrix is the Webbase matrix, 1,000,005 by 1,000,005 elements, which is a subset of a large world-wide web connectivity matrix. Figure 6.1 shows a visualization of the sparsity pattern for each matrix. All of the matrices are publicly available online from the University of Florida Sparse Matrix Collection [76].

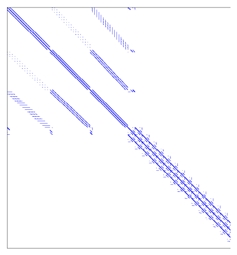

(a) Dense

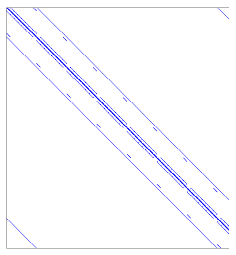

(b) Protein

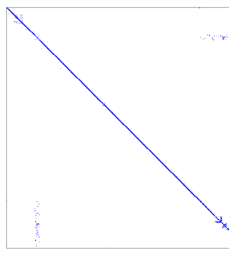

(c) LP

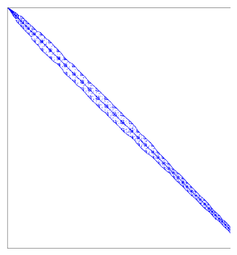

(d) FEM/Harbor

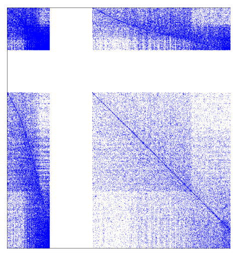

(e) FEM/Cantilever

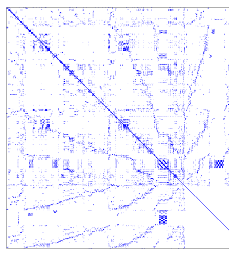

(f) FEM/Spheres

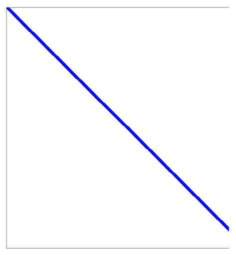

(g) QCD

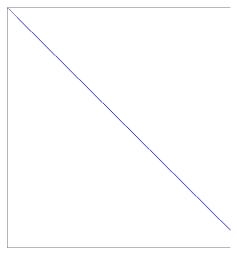

(h) Wind Tunnel


(i) FEM/Ship


(j) FEM/Accelerator


(k) Circuit


(l) Economics


(m) Epidemiology

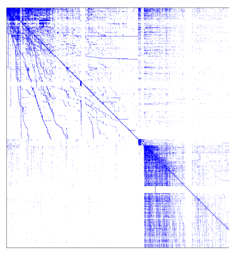

(n) Webbase

Figure 6.1: Matrix Test Data.

Table 6.1: Summary of unstructured matrices used for benchmarking performance (publically available from [76]).

Matrix	Rows	Columns	Nonzeros	Nonzeros/Row	Density
Dense	2,000	2,000	4,000,000	2,000	100.00000%
Protein	36,417	36,417	4,344,765	119.31	0.32761%
LP	4,284	1,092,610	11,279,748	2,632.99	0.24098%
FEM/Harbor	46,835	46,835	2,374,001	50.69	0.10823%
FEM/Cantilever	62,451	62,451	4,007,383	64.17	0.10275%
FEM/Spheres	83,334	83,334	6,010,480	72.13	0.08655%
QCD	49,152	49,152	1,916,928	39.00	0.07935%
Wind Tunnel	217,918	217,918	11,524,432	52.88	0.02427%
FEM/Ship	140,874	140,874	3,568,176	25.33	0.01798%
FEM/Accelerator	121,192	121,192	2,624,331	21.65	0.01787%
Circuit	170,998	170,998	958,936	5.61	0.00328%
Economics	206,500	206,500	1,273,389	6.17	0.00299%
Epidemiology	525,825	525,825	2,100,225	3.99	0.00076%
Webbase	1,000,005	1,000,005	3,105,536	3.11	0.00031%

Table 6.2: Data Sets* used for MCL and PageRank

Name	Type	# Nodes	# Interactions	Density
Yeast1	Protein	2,114	4,480	0.10%
Yeast2	Protein	2,361	13,828	0.25%
Human1	Genetic	22,283	24,669,643	4.97%
Human2	Genetic	14,340	18,068,388	8.79%
Mouse	Genetic	45,101	28,967,291	1.42%

*Protein-Protein and Genetic Interaction Data Sets from [77, 78, 79]. Publicly available online from [76].

6.1.2 Bioinformatics Algorithms

6.1.2.1 MCL and PageRank

Two protein-protein interaction data sets (*Yeast1* [77] and *Yeast2* [78]) and three genetic data sets (*Human1*, *Human2*, and *Mouse* [79]) are used for the MCL and PageRank benchmarks. The data sets were obtained from the Florida Matrix Market [76]. Their details are shown in Table 6.2.

Table 6.3: Synthetic Data Sets used for Compressive Sensing

Name	Rows	Columns	# Nonzeros	Density
CS1	400	1,600	161,206	25.19%
CS2	400	8,000	160,600	5.02%
CS3	2,500	10,000	6,293,543	25.17%
CS4	2,500	50,000	6,274,775	5.02%

Table 6.4: Test Platforms

	i7-2600	i7-4770	E5-2667	GTX 660	GTX TITAN	SX95T
Platform	CPU	CPU	CPU	GPU	GPU	FPGA
TDP [W]	95	84	130	140	250	25
Tech. Node [nm]	32	22	32	28	28	65
# of Cores	4/8*	4/8*	6/12*	960	2688	64
Clock Freq.	3.4GHz	3.9GHz	2.9GHz	980MHz	837MHz	150MHz
Perf. [GFLOP/s]	108.8	217.6	168	1,881.6	4,500	19.2
Memory [GB]	16	32	256	2	6	2.25
Mem. BW [GB/s]	21	25.6	51.2	144.2	288.4	35.75
BLAS Library	MKL 11.0	MKL 11.0	MKL 11.0	cuSPARSE 5.0	cuSPARSE 5.0	This Work

*physical/virtual cores

6.1.2.2 Compressive Sensing

Several random sparse sensing matrices (Φ) were generated using the `sprand()` function in MATLAB 2013a for the compressive sensing benchmarks. Table 6.3 details the statistics of each sensing matrix. *CS1* and *CS2* correspond to a 20 by 20 DNA microarray with a 25% and 5% sparsity ratio, respectively. Similarly, *CS3* and *CS4* correspond to a 50 by 50 DNA microarray with 25% and 5% sparsity ratios.

6.1.2.3 Tomographic Reconstructions

A 3D Shepp-Logan phantom was used to generate synthetic data in MATLAB 2013a for the SIRT benchmarks. Projections images from an array of 2048×2048 detectors with 128 projection angles were used to reproduce volumes of 512^3, 1024^3, and 2048^3.

6.2 Hardware and Software Test Platforms

In total, three CPUs, two GPUs, and one FPGA were used in a total of four different computing platforms. The sparse subroutines from the Intel Math Kernel Library (MKL) version 11.0 [26] were used in the CPU benchmarks. The sparse subroutines of the NVIDIA cuSPARSE CUDA library [28] version 5.0 were used for the GPU benchmarks. The FPGA system was designed using Xilinx ISE 11.5 and controlled via MATLAB 2013a. Table 6.4 summarizes the specifications of the test setups.

The first computer, representative of a mid-range desktop computer, is a 64-bit Linux desktop equipped with 16GB of memory, an Intel Core i7-2600 processor (4 physical cores with hyper-threading, for a total of 8 virtual cores), and an NVIDIA GeForce GTX 660 graphics card (960 CUDA cores). The second computer, characteristic of a high-end desktop computer, is a 64-bit Windows desktop with 32GB of memory, an Intel Core i7-4770 processor (4 physical cores with hyper-threading, for a total of 8 virtual cores), and an NVIDIA GeForce GTX TITAN graphics card (2688 CUDA cores). The third computer, a 64-bit Linux server with an Intel Xeon E5-2667 processor (6 physical cores with hyper-threading, for a total of 12 virtual cores), is typical of a single node in a HPC research cluster.

The sparse linear algebra kernel was evaluated on the open-source academic research platform "ROACH" (Reconfigurable Open Architecture Computing Hardware) [73]. The "ROACH" platform is equipped with a Virtex-5 SX95T FPGA, a PowerPC running Linux, two 36Mb QDRII+ SRAMs, 2GB of error correcting DDR2 SDRAM, two gigabit Ethernet ports, four 10 gigabit Ethernet ports, and two high-speed ZDOK+ connectors. The system has a total for a combined peak memory bandwidth of 35.75 GB/s. The PowerPC allows a computer running MATLAB to interface with "software registers," BRAMs, and FIFOs on the FPGA, as well as to load data in and out of the board-level QDRs and DRAM (Figure 3.7). The FPGA kernel implements 64 single-precision floating-point processing elements running at 150MHz for a peak performance of 19.2 GFLOP/s.

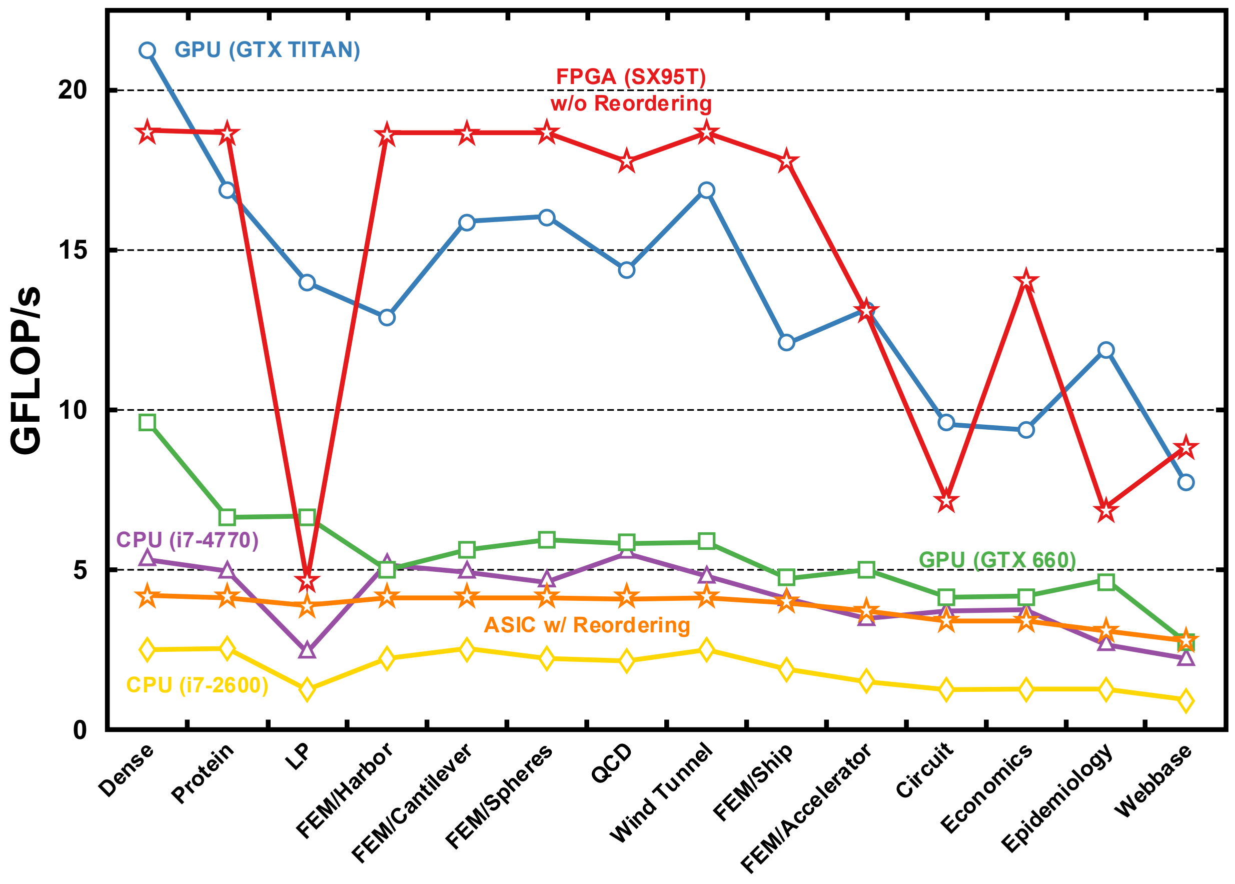


Figure 6.2: Raw computational performance of the CPU, GPU, FPGA, and ASIC sparse-BLAS kernels.

6.3 Performance Results

6.3.1 Sparse Matrix Collection

Figure 6.2 compares the raw computational performance (in GFLOP/s) of the CPU, GPU, FPGA, and ASIC SpMxV kernels for all of the matrices tested. SpMxV on the two CPUs showed a performance drop of 20-50% compared to dense matrices, while the two GPUs showed a performance drop of 30-60%. Figure 6.3 compares the computational efficiency of the CPU, GPU, and FPGA SpMxV kernels for all of the matrices tested. For a memory bound algorithm like SpMxV, the computational efficiency is strongly determined by the

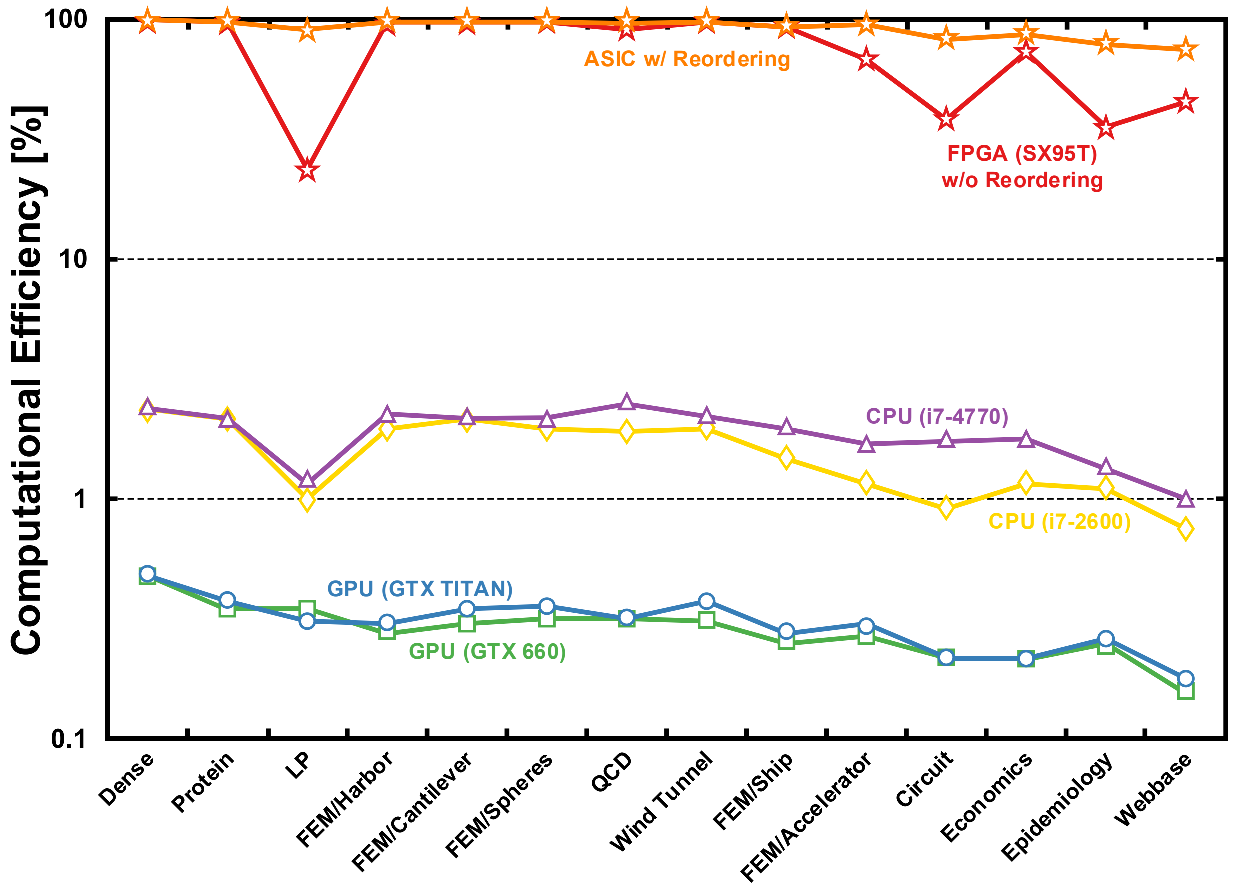


Figure 6.3: Computational efficiency of the CPU, GPU, FPGA, and ASIC sparse-BLAS kernels.

memory hierarchy (i.e. the cache structure and size). The computational efficiency is calculated as the ratio of the measured SpMxV performance, in GFLOP/s, over the theoretical peak GFLOP/s achievable for each platform.

The Core i7-2600 and Core i7-4770 processors achieved an average performance of 1.72 and 4.08 GFLOP/s, respectively, across all 14 test matrices. The resulting computational efficiencies were 1.58% and 1.88%. Overall, the computational efficiency of both CPUs was 1-2% for all of the test matrices. The Core i7-4770 processor was able to achieve a 2.37x speedup over the Core i7-2600, despite only having 22% more memory bandwidth, due to its more efficient memory accesses with its larger vector processing cores.

The largest drops in performance for the CPUs were recorded using the sparsest and most irregular matrices: LP, Circuit, and Webbase. These matrices had a significantly higher rate of cache misses due to their large size and overall sparsity. The relativity small number of nonzero elements per row (especially for the Webbase matrix) also added significant overhead by having to flush the pipeline more often. The very asymmetric nature of the LP matrix was also hard for the CPU architectures to handle.

Similarly, the GTX 660 and GTX Titan GPUs achieved an average performance of 5.42 and 13.71 GFLOP/s, respectively across all 10 matrices. The resulting computational efficiencies were 0.29% and 0.30%. Overall, the computational efficiency of both GPUs was between 0.2-0.5% for all of the test matrices. The GTX Titan had an average speed up of 2.53x over the GTX 660 GPU, which is consistent with the GTX Titan having 2x the memory bandwidth and 2.39x the number of processing cores as the GTX 660. Because individual process threads are organized differently on the GPU, each CUDA core had to be flushed far less often than the CPU for the LP, Circuit, and Webbase matrices, leading to more even performance. However, the Webbase matrix showed the worst performance for both GPUs.

On the Linux test setup, the GTX 660 had an average speedup of 3.15x over the i7-2600 processor, and on the Windows setup, the GTX Titan had a 3.36x speedup over the i7-4770 processor. Despite the roughly 3x performance increase by the GPUs (Figure 6.2), the CPUs are about 6x more computationally efficient than the GPUs (Figure 6.3). The GPUs use 100-300x more processing cores to achieve a total, theoretical peak performance roughly 20x greater than that of the CPU, but only have about 8x more memory bandwidth. The cache structure of a GPU is smaller and has higher latency that of a CPU [80,81]. GPUs are designed to mask random memory accesses for computationally intensive algorithms, leading to much larger penalties in efficiency for cache misses when compared to a CPU.

The SpMxV kernel running on the Virtex-5 SX95T FPGA achieved an average performance of 14.56 GFLOP/s, for a computational efficiency of 75.82%, across all 14 matrices. The Dense matrix achieved a peak performance of 19.16 GFLOP/s for a computational efficiency of 99.8%. This performance represents an average speedup of 8.47x and 3.57x over the i7-2600 and i7-4770 CPUs and a 2.69x and 1.06x speedup over the GTX 660 and GTX Titan GPUs. Moreover, the computational efficiency of the FPGA SpMxV kernel had an average improvement of 40x and 252x over the CPUs and GPUs, respectively. The biggest drops performance were the LP, Circuit, Epidemiology, and Webbase matrices, averaging only 35.74%, while the other 10 matrices averaged 91.85%. This is due to two factors: (1) the extremely short column length of the LP matrix, and (2) the very sparse nature of the Circuit, Epidemiology, and Webbase matrices. This led to a large amount of control overhead for the LP matrix and a large number of data hazards for the other matrices.

The sparse-BLAS kernel running on the ASIC achieved an average performance of 3.49 GFLOP/s, for a computational efficiency of 90.94%, across all 14 matrices. We were able to boost the average computational efficiency of the LP, Circuit, Epidemiology, and Webbase matrices to 82.13% with the use of the "Shuffler." We were also able to boost the average efficiency of other 10 matrices to 95.29%. This represents a 20% boost in average computational efficiency over the more simplistic method of stalling to resolve data hazards (with a 400% improvement in efficiency for the 4 worst performing matrices). Overall, the computational efficiency of the ASIC kernel had an average improvement of 48x and 303x over the CPUs and GPUs, respectively.

6.3.2 Bioinformatics Algorithms

Figure 6.4 shows the normalized execution time for the MCL benchmark on each of the test platforms. An expansion power of $p = 2$, an inflation parameter of $r = 2$, and a chaos threshold of $e = 10^{-3}$ was used for each data set. The algorithm converged in 13 iterations for the *Yeast1* data set, 18 iterations for *Yeast2*, 20 iterations for *Human1*, 14 iterations for *Human2*, and 21 iterations for the *Mouse* data set. The speedup of the FPGA over the CPUs and GPUS for the *Yeast1* and *Yeast2* was rather modest: only 2-4x due to the relatively small size of the problems. However, the speedups for the genetic data sets increased to 10-18x for the i7-2600 and i7-4770 desktop CPUs. The substantially larger amount of system memory greatly improved the performance of the E5-2667 processor, limiting the FPGA

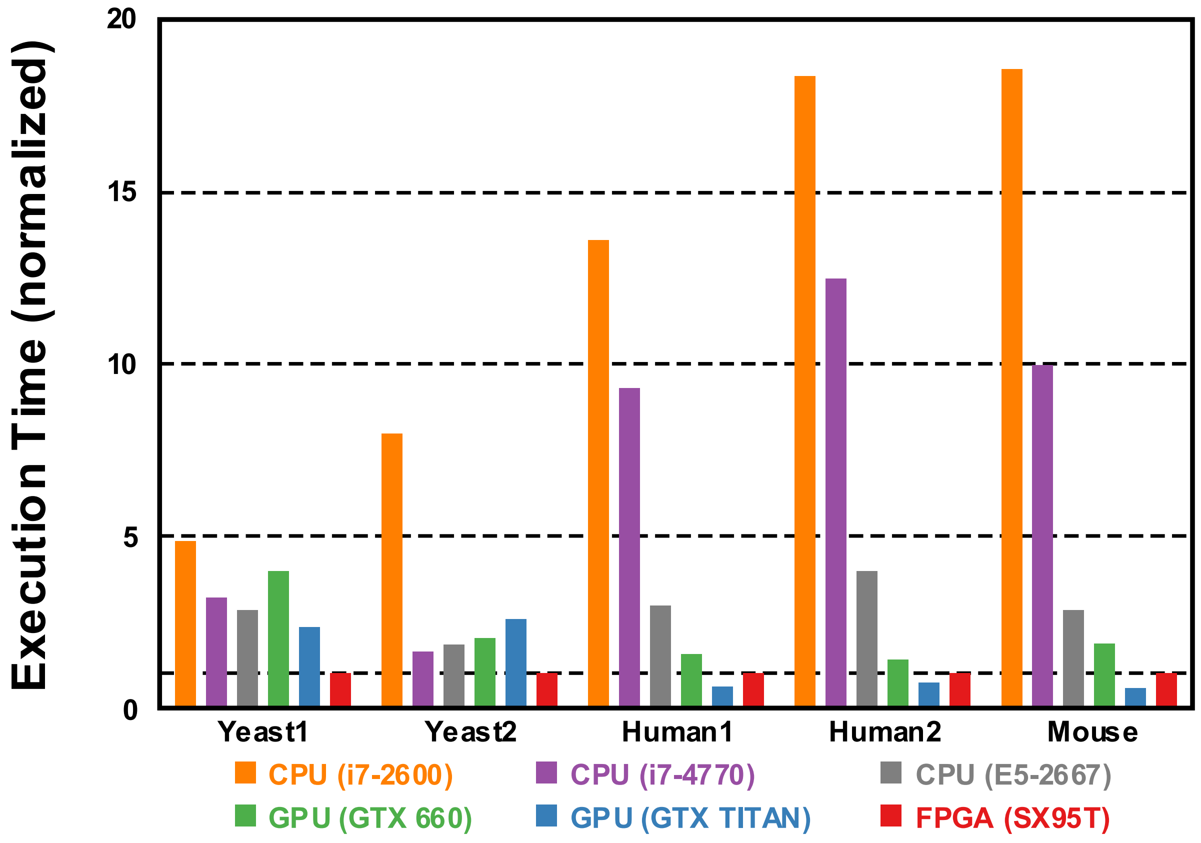


Figure 6.4: Normalized execution time for MCL on each of the test platforms from Table 6.4 using the data sets from Table 6.2.

speedup to 3-4x. For the larger data sets, the superior memory bandwidth of the GPUs allowed them to excel at the SpMxM computations of the MCL algorithm. In particular, the GTX TITAN showed a 1.5-2x speedup over the "ROACH" FPGA system. However, this result is to be expected: the GTX TITAN has 42 times the number of processing cores than the FPGA, with each core running 5.5 times faster with 8 times the memory bandwidth.

The results of the PageRank algorithm, the first of the SpMxV benchmarks, is summarized in Figure 6.5. For the smaller datasets, the GPUs preformed poorly, with an FPGA speedup of 16x and 8x for the *Yeast1* and *Yeast2* datasets respectively. The small problem sizes simply did not allow the GPUs to leverage all of their available processing cores to mask the very high memory latency of their GDDR5 memory. However, for the larger problem sizes, the GPUs were more effective at hiding this latency, with the GTX TITAN showing a 1-1.2x speedup over the FPGA. The Sandy-Bridge architecture of the i7-2600 and E5-2667

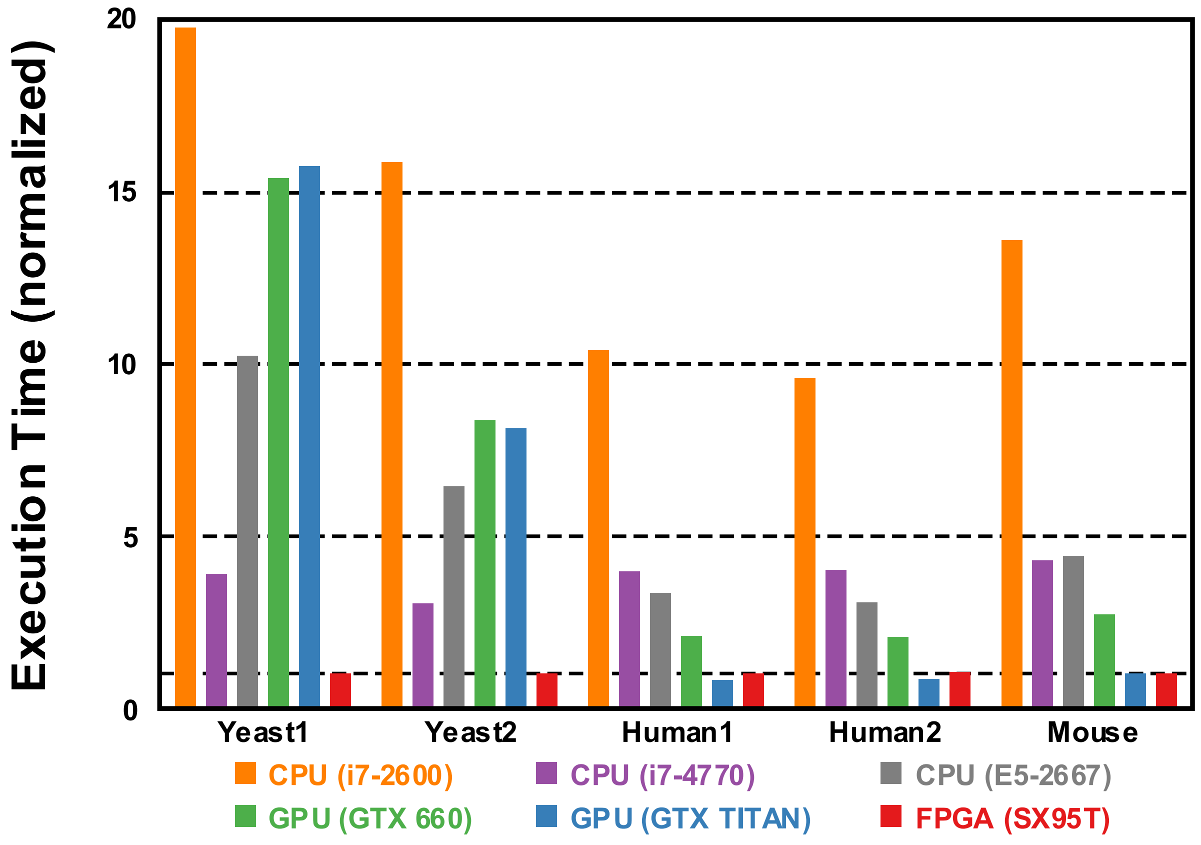


Figure 6.5: Normalized execution time for the PageRank algorithm on each of the test platforms from Table 6.4 using the data sets from Table 6.2.

processors appears to have severely limited their SpMxV performance, because the Haswell microarchitecture of the i7-4770 CPU was relatively constant across all of the datasets. On average, the CPUs were 5-20 times slower than the FPGA for the PageRank benchmarks.

The normalized execution time for the compressive sensing reconstructions is shown in Figure 6.6. Due to the complexity of the reconstruction algorithm, speedups were more modest compared to other SpMxV algorithms. For the smaller datasets, *CS1* and *CS2*, the FPGA was consistently 3 to 6 times faster, with a slightly larger speedup for the sparse dataset *CS2*. For the larger datasets, *CS3* and *CS4*, the CPUs were roughly 6 to 12 times slower. The FPGA showed a 2x to 2.5x speedup over the GTX 660 GPU for the *CS3* and *CS4* datasets, respectively. While the GTX TITAN showed a 1.12x and a 1.02x speedup over the FPGA.

The normalized execution time for the 3D SIRT tomographic reconstructions is shown

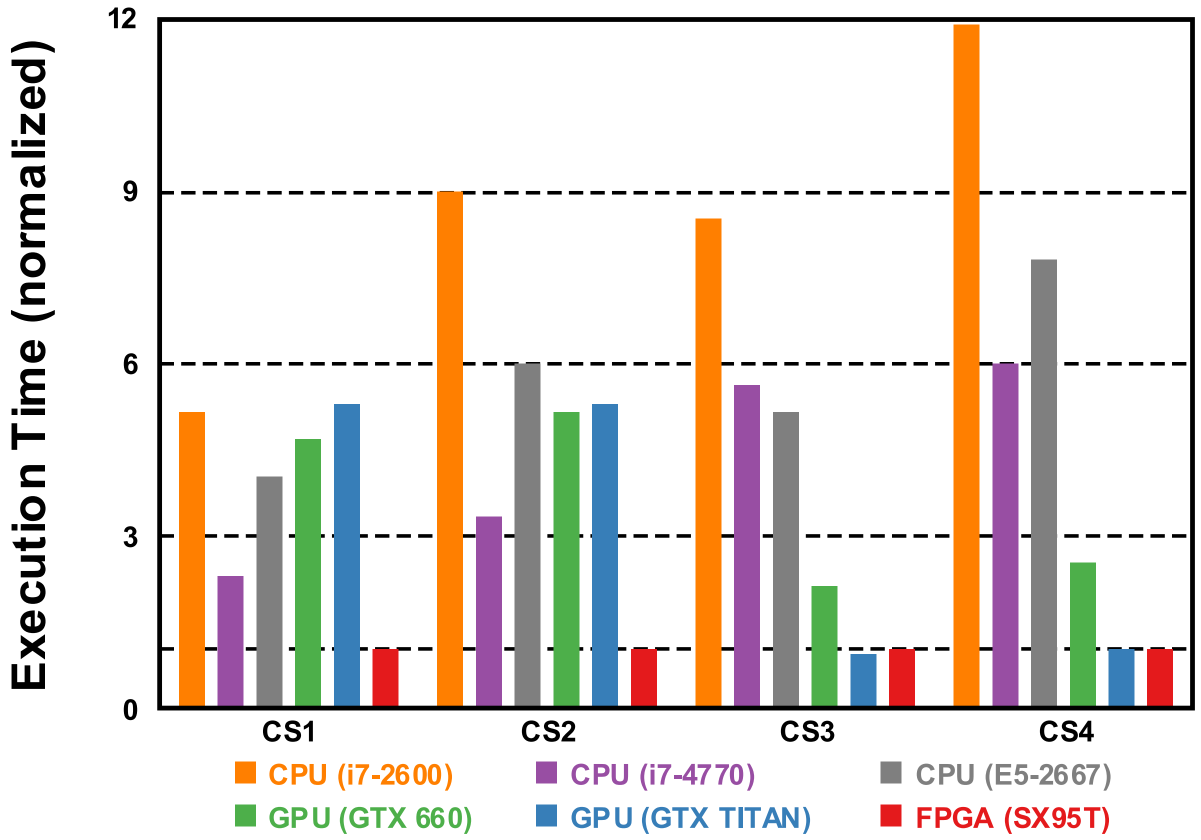


Figure 6.6: Normalized execution time for compressive sensing reconstructions on each of the test platforms from Table 6.4 using the synthetically generated data sets from Table 6.3.

in Figure 6.7. It should be noted that the FPGA sparse linear algebra kernel managed to outperform both the CPUs and the GPUs for all of the problem sizes. For the largest problem size (2048^3), the “ROACH” system showed a 4x speedup over the GTX TITAN and a 25x speedup over the GTX 660. The 2GB of memory on the GTX 660 appears to have been the limiting factor for the tomographic reconstructions. Again, the Haswell microarchitecture of the i7-4770 CPU appears to be much better suited for the SpMxV operations that dominate the SIRT algorithm (with 2 SpMxV per iteration). As expected, the i7-2600 performed the worst and was almost 50 times slower than the FPGA for the largest reconstruction problem.

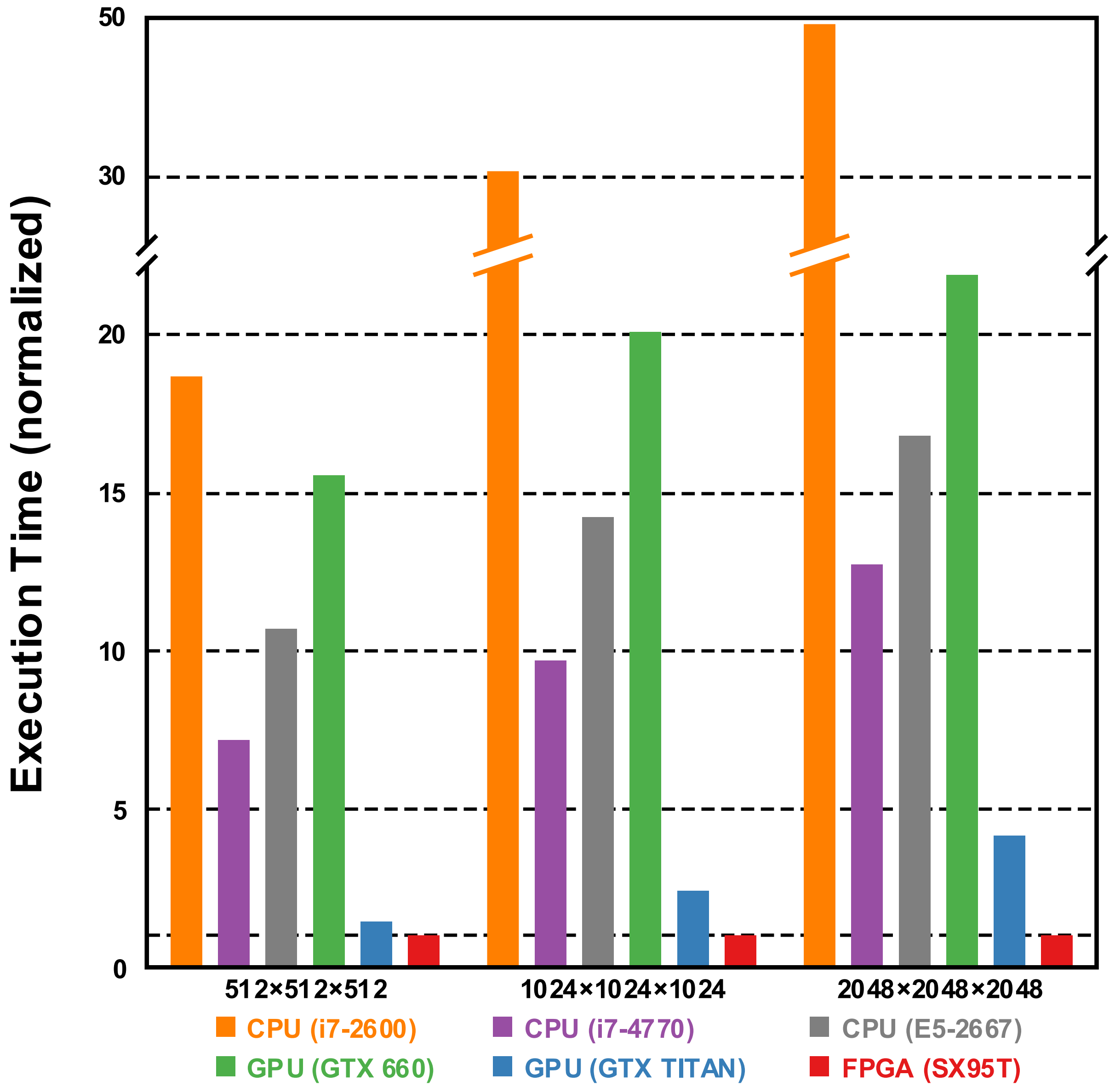


Figure 6.7: Normalized execution time for preforming SIRT on each of the test platforms from Table 6.4 for 3 different resolutions using synthetically generated data in MATLAB from a 3D SheppLogan phantom.

6.4 Energy Efficiency

6.4.1 FPGA SpMxV Kernel

The average power consumption of the i7-2600 and i7-4770 processors was measured to be 77.2W and 66.3W, respectively. The resulting power efficiencies are 22.3 MFLOP/s/W and

Table 6.5: Energy Efficiency

Platform	[W]	[GFLOP/s]	MFLOP/s/W
i7-2600	77.2	2.01	26 (133.1x)
i7-4770	66.3	4.59	69 (50.1x)
E5-2667	103.7	3.42	33 (104.8x)
GTX 660	99.0	5.79	58 (59.7x)
GTX TITAN	163.0	14.86	91 (38.0x)
SX95T	5.1	17.64	3,460 (1x)

61.9 MFLOP/s/W. Similarly, the average power of the GTX 660 and GTX Titan were measured to be 99W and 163W, respectively. The resulting power efficiencies are 54.8 MFLOP/s/W and 84.1 MFLOP/s/W. The worst case power of the SX95T FPGA was measured to be 5.1 W, resulting in a power efficiency of 2,854 MFLOP/s/W. This represents more than a 46x and 34x improvement in energy efficiency over the CPU and GPU implementations, respectively.

6.4.2 FPGA Sparse-BLAS Kernel

Power and memory utilization are the two most important concerns for building scalable, HPC, cloud-based computing environment for memory-bounded algorithms. To compare the average energy efficiency of each platform, we measured the peak, sustained performance (in GFLOP/s) and average power consumption for the SIRT tomographic reconstruction and PageRank benchmarks. The i7-2600, i7-4770, and E5-2667 processors achieved an average of 2.01, 4.59, and 3.42 GFLOP/s, with an average power consumption measured to be 77.2W, 66.3W, and 103.7W, respectively. The resulting power efficiencies are 26MFLOP/s/W, 69MFLOP/s/W, and 33MFLOP/s/W. Similarly, the average power of the GTX 660 and GTX TITAN were measured to be 99W and 163W for a peak performance of 5.79 and 14.86 GFLOP/s, respectively. The resulting power efficiencies are 58MFLOP/s/W and 91MFLOP/s/W. The average power of the SX95T FPGA was measured to be 5.1W for a peak performance of 17.64 GFLOP/s, resulting in a power efficiency of 3,460 MFLOP/s/W. This represents more than a 50x and 38x improvement in energy efficiency over the CPU

Table 6.6: Memory Bandwidth Utilization

Platform	**[GB/s]**	**Peak [%]**
i7-2600	9.85	46.9
i7-4770	21.57	84.3
E5-2667	14.7	28.7
GTX 660	71.22	49.4
GTX TITAN	166.43	57.7
SX95T	35.28	98.9

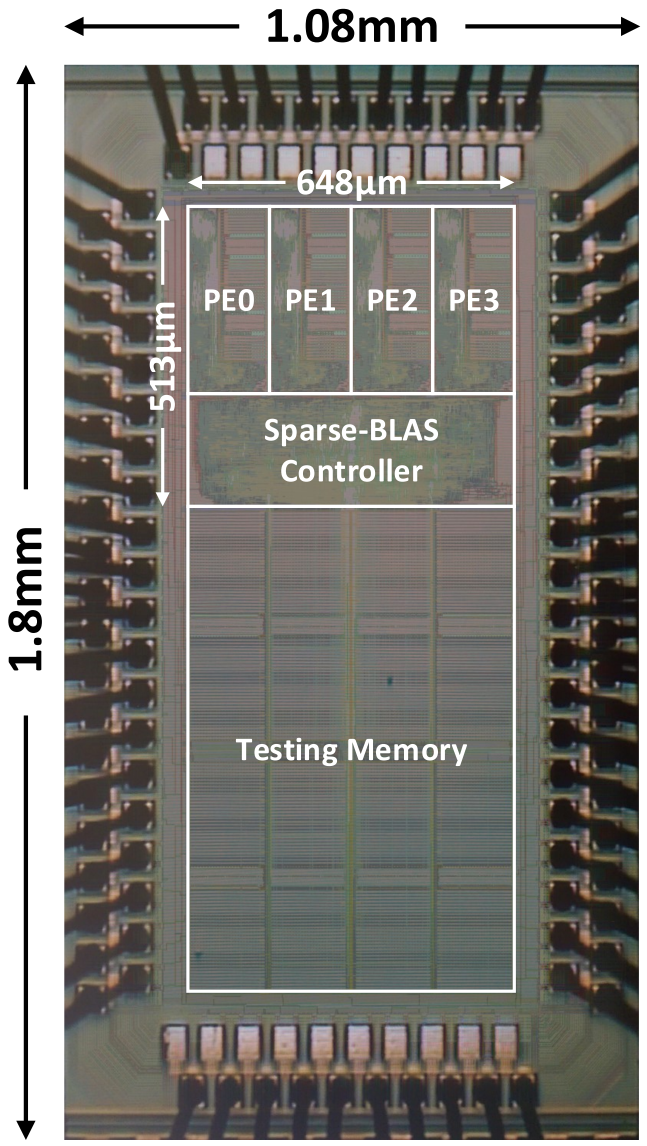
and GPU implementations, respectively. Table 6.5 summarizes the measured power, performance, and energy efficiency for each platform.

For the memory bandwidth utilization of the SIRT tomographic reconstruction and PageRank benchmarks, we could directly measure the average memory bandwidth (in GB/s) used by the GPUs and the FPGA. However, we had to estimate the bandwidth utilization from the reported values of the cache- and page-miss counters of the CPUs. For the CPUs, the sparse algorithms averaged between 30% to 85% of their theoretical peak memory bandwidth, while the GPUs averaged between 50% to 60%. The FPGA managed to average 98.9% of its theoretical peak memory bandwidth. The FPGA-based sparse-BLAS kernel showed a 2.4x improvement in bytes/FLOP (i.e. computation per memory access) compared to the i7-4770 CPU and a 5.6x improvement in bytes/FLOP over the GTX TITAN GPU. This result shows that regularizing the memory accesses and minimizing the communication costs in the FPGA kernel can led to large performance and energy gains. Table 6.6 summarizes the memory bandwidth utilization for each platform.

6.4.3 ASIC Sparse-BLAS Kernel

Figure 6.8 shows the chip micrograph of the sparse-BLAS ASIC which was fabricated in a 40nm 1P10M CMOS process. With a core area of 513μm by 648μm and almost 7 million transistors, the chip can operate up to 515MHz with a core voltage between 0.55V and 1V. Running at 515MHz, the four PEs can achieve a maximum performance of 4.12 GFLOP/s.

A shmoo plot of the operating frequency vs. core voltage for the sparse-BLAS ASIC



Technology	40nm 1P10M CMOS FO4 16.3ps (TT)
Core V_{DD}	0.55 to 1V
Frequency	≤ 515 MHz
Energy Efficiency	≤ 190 GFLOP/s/W
I/Os	Digital: 60 Power: 60
I/O V_{DD}	1.8V
Core Size	513µm × 648µm
Transistor Count	6.98 million

Figure 6.8: Sparse-BLAS chip micrograph.

chip can bee seen in Figure 6.9. In the shmoo plot, red indicates failure and green indicates passing. Figure 6.10 shows a plot of measured power vs. operating frequency for the sparse-BLAS ASIC chip. The minimum energy point (MEP) occurs with a core voltage of 0.6V at an operating frequency of 160MHz, with a measured power of 6.73mW. The resulting energy efficiency is 190.31 GFLOP/s/W. This represents more than a 3,073x, 2,262x, and 66.6x improvement in energy efficiency over the CPU, GPU, and FPGA implementations, respectively.

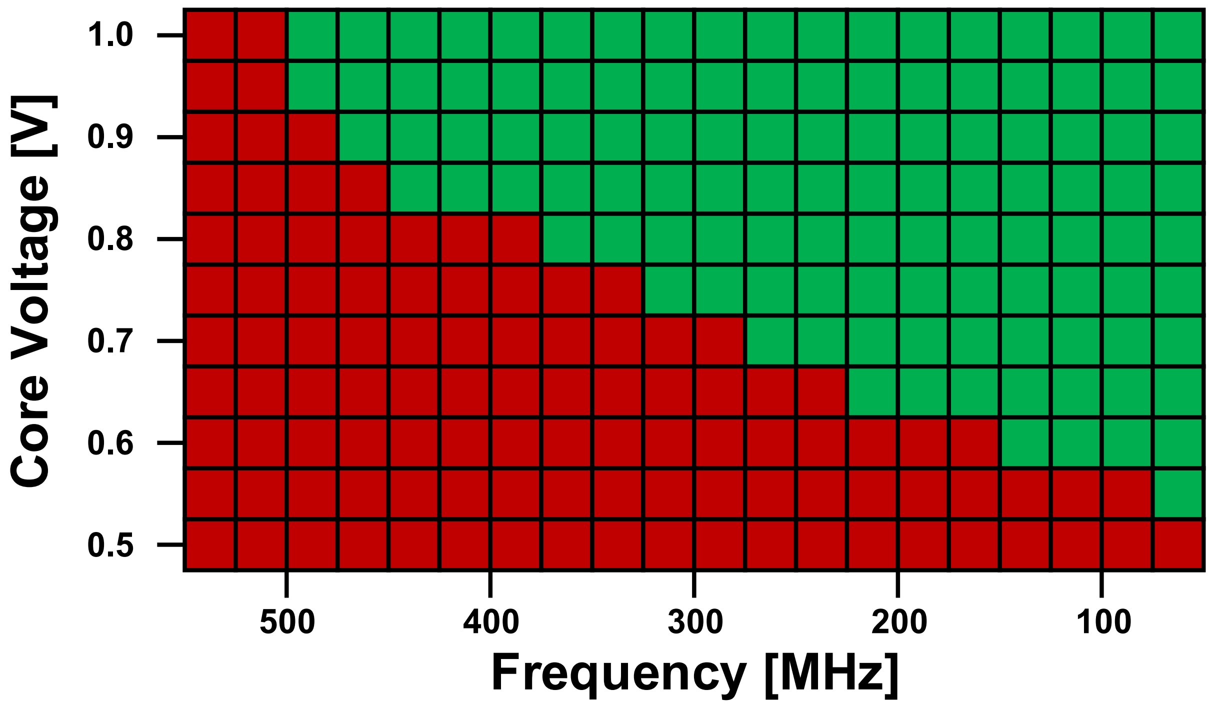


Figure 6.9: Shmoo plot of operating frequency vs. core voltage for the sparse-BLAS ASIC chip. Red indicates failure and green indicates passing.

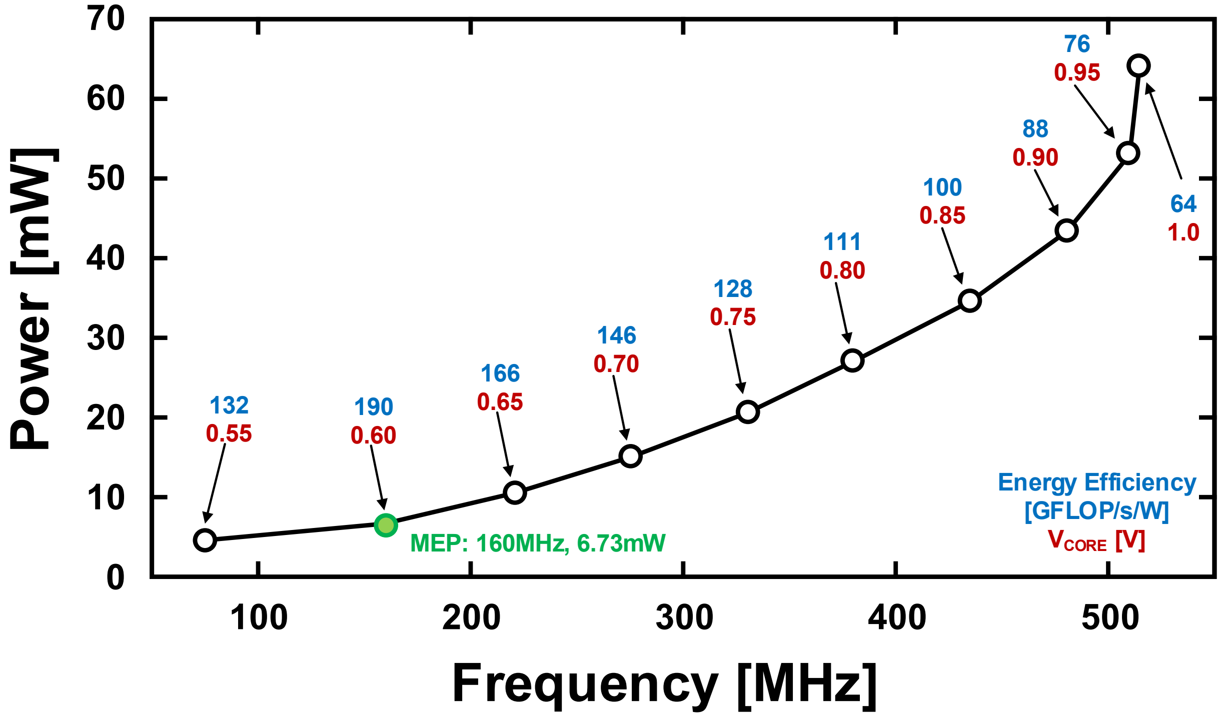


Figure 6.10: Chip power vs. operating frequency for the sparse-BLAS ASIC chip.

CHAPTER 7

Magnetic Tunnel Junctions

The focus of this chapter is to introduce the MTJ device, and the field of spintronics, specifically focusing on its applications in memories (STT-MRAM and MeRAM). The first section provides a brief background on spintronics—its history and fundamental physical operation. Alternative devices (e.g. spin FETs, MBTs, and spin LEDs) and applications are also discussed. The second section covers the characteristics and unique properties of the MTJ device, while the third section introduces various different types of spintronic memories. The fourth section presents a Verilog-A model capable of capturing these behaviors and simulating these spintronic memories. The fifth, and final section of this chapter, verifies this model with qualitative and quantitative comparisons to measured devices and detailed micromagnetic simulations.

7.1 Introduction to Spintronics

Spintronics, the amalgamation of the words "spin" and "electronics," involves the active control and manipulation of electron spin in solid-state electronics [82]. In traditional electronic devices, information processing works on the principle of control over the flow of charge through a semiconductor material. Large scale, non-volatile memories (e.g., hard disk drives or HDDs) exploit ferromagnetism to store information by forcing the spin alignment of many electrons [83]. Spintronics, as a whole, aims to merge information processing and storage through the use of spin-polarized currents [4].

7.1.1 History

Early work into spintronics began in the mid-1930s with the discovery of unusual resistance behavior in ferromagnetic materials at extremely low temperatures [82]. Electron tunneling measurements played a key role in early experimental work, with several key experiments in the early 1970s demonstrating the viability of spin filters (discussed later). In 1975, Jullière [84] formulated his now-famous conductance model describing the change of conductance between the parallel and antiparallel states of an MTJ. However, it wasn't until

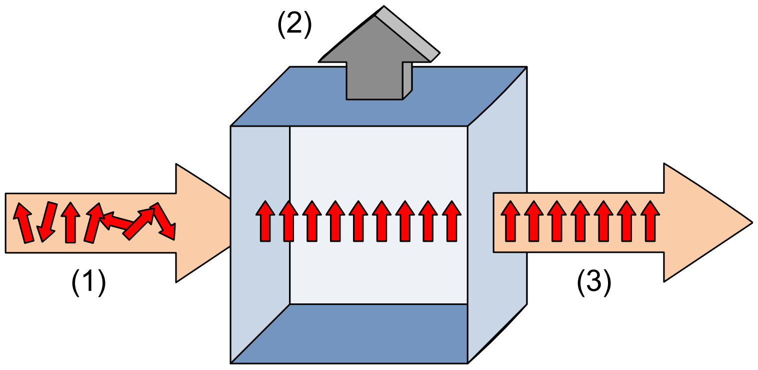


Figure 7.1: A cartoon of the operation of a spin polarizer. A spin unpolarized current enters at (1). It is then spin polarized in the direction of (2), before exiting the spin polarizer at (3).

the mid-to-late 1980s that the room temperature magnetoresistive effects were discovered. Anisotropic magnetoresistive (AMR) layers were first used to construct AMR-MRAM to replace bulky and heavy plated-wire radiation-hard memories [4]. AMR was quickly replaced by the discovery of giant magnetoresistance (GMR) in 1988 [7]. Since the discovory of GMR, electron spin has formed the basis of almost all electronic information storage [85].

In the early 1990s, MTJ materials with higher TMRs (on the order of 20% at room temperature) were discovered [4]. Since then, MTJ structures (using MgO insulating barriers) with TMRs on the order of 1000% have been demonstrated at room temperature [86]. Within ten years of its discovery, spintronics has grown into a billion dollar industry, with commercial sales exceeding $3 billion in 2005 [7]. Despite these successes, spin injection from ferromagnetic layers into semiconductors remains a significant bottleneck in semiconductor-based spintronics. Recently, much emphasis has been placed in trying to induce ferromagnetism in a semiconductors to produce dilute magnetic semiconductors (DMS) [85]. DMS has the potential to improve the Curie temperature and magnetic band gap of future spintronic devices [87].

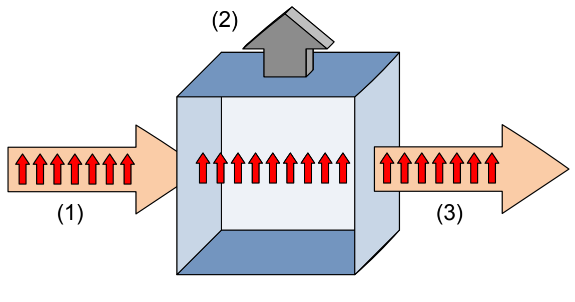


Figure 7.2: A cartoon of the operation of a spin filter to a (a) parallel and (b) an antiparallel current. A spin-polarized current enters at (1) and filters in the direction of (2), before exiting the spin polarizer at (3).

7.1.2 Principle of Operation

Electron spin is a "pseudovector" with a fixed magnitude but a variable direction (spin polarization). The spin polarization of an electron can be made bistable by placing it in a magnetic field. In the presence of a magnetic field, only spin polarizations parallel or antiparallel to the field are possible [4]. This property introduces the concept of a spin polarizer (Figure 7.1). A thin ferromagnetic layer can act as a spin polarizer. When a spin unpolarized current passes through the ferromagnetic layer, it tends to become spin-polarized in the direction of magnetization [7]. Another key aspect to spintronics is the concept of a spin filter (Figure 7.2). A spin filter will only pass a current if the two are polarized in the same direction. If the current and filter are completely antiparallel, no current is passed. Ferromagnetic films also display the properties of a spin filter [7]. A "spin valve" can be constructed by using a spin polarizer in conjunction with a spin filter [83]. By controlling the angle of magnetization between the polarizer and the filter, a magnetically controlled spin valve can be formed. The spin valve effect is exploited in MRAMs to use MTJs as the memory storage element [4].

7.1.3 Other Devices and Applications

Several kinds of "spin transistors" exist, including the spin field-effect transistor (spin FET), the magnetic bipolar transistor (MBT), and hot-electron spin transistors [82, 85]. Structurally similar to a MOSFET, a spin FET sandwiches the conducting channel between two ferromagnetic layers. When the ferromagnets are aligned in the parallel configuration, the spin FET behaves like a normal MOSFET. However, when configured in the antiparallel alignment, transistor will be shut off [88]. Spin FETs can be easily integrated into existing CMOS circuitry and provide much larger ON/OFF current ratios [82]. MBTs are essentially BJTs with the addition of a ferromagnetic spin injector attached to the emitter. In an MBT, the gain factor β heavily depends upon nonequilibrium spin polarization and is called magnetoamplification [89]. MBTs can be used to generate almost 100% spin-coherent currents that can be very long lived [82, 89].

Another potential application of spintronics is optics, specifically, through the use of spin light emitting diodes (spin LEDs) and spin selective Kerr rotators [85]. In a spin LED, the polarization of the light emitted is modulated through the application of an external magnetic field [90]. Variable polarized LEDs promise to provide more energy efficient displays and significantly higher signal-to-noise ratio (SNR) in optical communications [85]. A Kerr rotator takes advantage of the magneto-optic Kerr effect (MOKE), the unique optically-reflective properties of magnetic materials, to manipulate the polarity of reflected light. Traditionally, Kerr rotators have many applications in the microscopic imaging of magnetic domains, magnetic media, and terahertz lasers [91]. A spin selective rotator, with the application of a bias voltage, can be made to reflect incident light either with or without a large Kerr rotation angle [85].

Spin-based quantum computing makes use of entangled quantum dots (qubits) [82]. A quantum dot, or a quantum point contact, is a device whose individual quantum states are freely controllable. Conceptually, a spin-based quantum dot contains a single electron whose spin orientation (up or down) is user controlled. By entangling several quantum dots together, a quantum computer can be constructed. Additionally, the boolean logic gates of a spin-based quantum computer can be constructed using with magnetic dots through the use of dipole-dipole interactions [4].

7.2 The Magnetic Tunnel Junction

This section is intended to describe the major device characteristics observed in MTJs. The science responsible for each effect, as well as their importance to the MTJ model, is discussed.

7.2.1 Resistance Hysteresis

The large resistance hysteresis present in MTJs makes them very well-suited as a non-volatile memory element. The source of this hysteresis is very nicely explained by the spin-valve structure of an MTJ [82]. As mentioned before in Figure 7.9, an MTJ consists of a thin insulating layer sandwiched between two ferromagnetic layers. The electromagnetic dynamics of the system allows for only two possible states: parallel or antiparallel [7]. The two ferromagnetic layers are magnetized in the same direction while in the parallel state and in the opposite directions while in the antiparallel state. When a current flows through the MTJ, one ferromagnetic layer acts as a spin polarizer and the other as a spin filter. In the parallel state, since the two ferromagnetic layers are aligned, the current is passed undisturbed, creating a low resistive state (R_P). However, in the antiparallel state, the spin filter will block the antiparallel current generated by the polarizing layer, creating a high resistive state (R_{AP}). Tunnel magnetoresistance (TMR) is a metric for determining the efficiency of spin-valve operation in an MTJ [8]. TMR is defined as:

$$TMR = \frac{R_{AP} - R_P}{R_P}. \tag{7.1}$$

Figure 7.3 shows a sample resistance hysteresis of an MTJ by sweeping the bias voltage. R_P and R_{AP} are clearly evident, along with several other characteristics to be discussed: critical switching currents, switching asymmetry, and the bias voltage dependence of TMR.

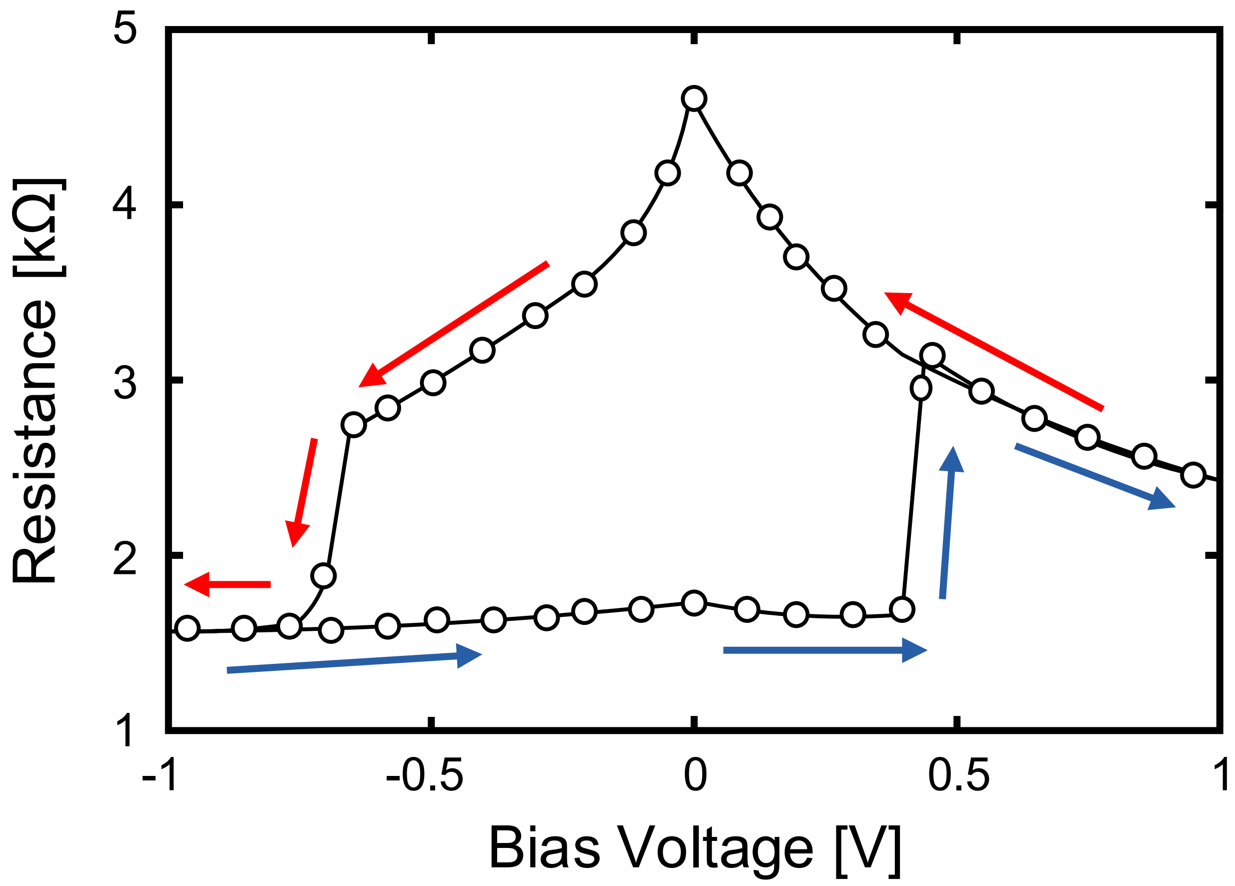


Figure 7.3: Resistance hysteresis of an MTJ. Switching from $P \rightarrow AP$ (blue arrows) and $AP \rightarrow P$ (red arrows).

7.2.2 Critical Switching Current

7.2.2.1 Asymmetric Switching Currents

It should be noted that the critical switching currents are asymmetric, with $I_C(P \rightarrow AP) > I_C(AP \rightarrow P)$ [92]. This effect was predicted by Slonczewski [9] with his discovery of the spin-torque transfer phenomena. This asymmetry is proportional to and increases linearly with TMR [93]. The simplest explanation of this behavior is that the antiparallel configuration is a lower energy state than the parallel case [82], making it is easier to switch to the antiparallel state than the parallel state. Several techniques exist to minimize the asymmetry. Lee et al. [94] were able to tune the magnetostatic offset field (using an external magnetic field) with exceptional results, reducing the asymmetric current ratio from 1.51 to 1.04. Yao et al. [95]

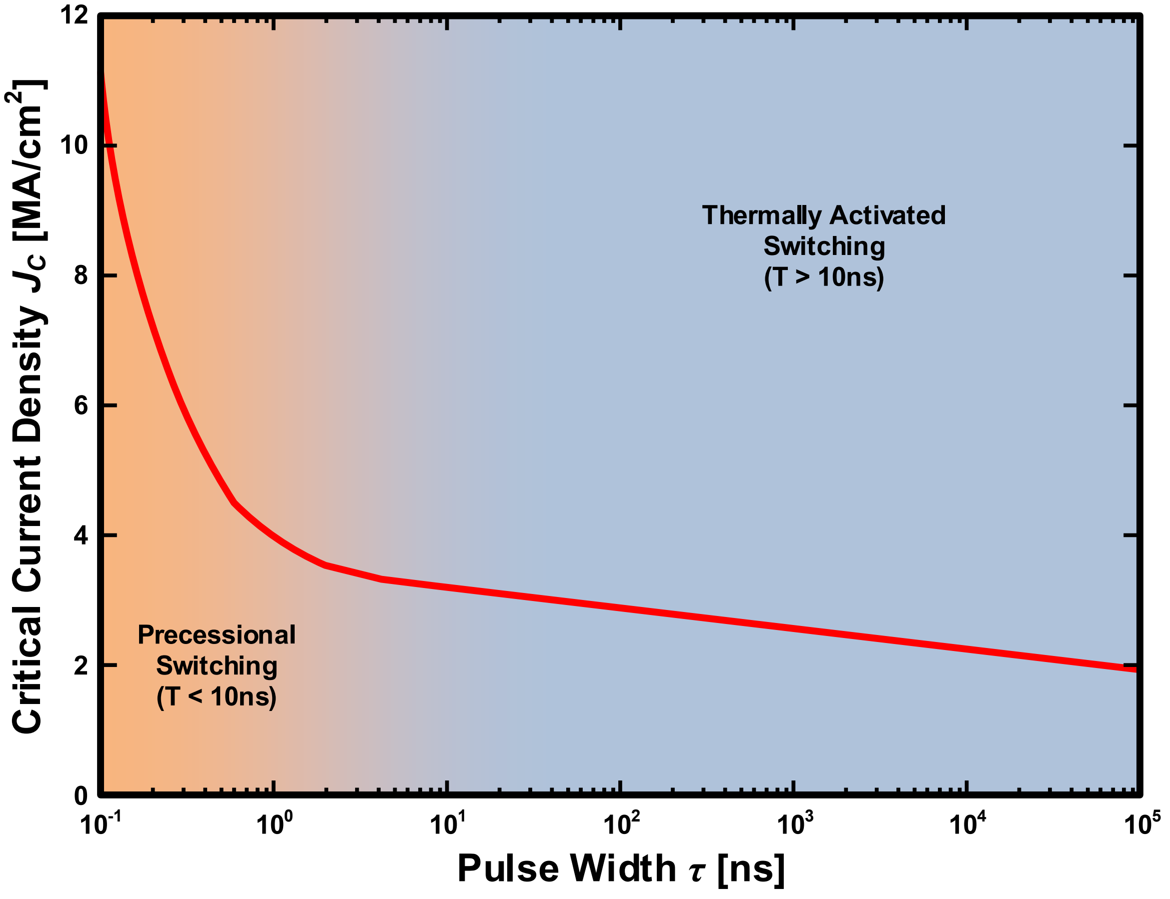


Figure 7.4: MTJ switching regimes.

were able to reduce the offset from 1.50 to 1.28 with the introduction of a nanocurrent-channel layer to the MTJ stack.

7.2.2.2 Switching Regimes

In MTJs, two types of magnetic switching occur due to spin-torque transfer: precessional and thermally activated switching [96, 97]. Precessional switching occurs on a nanosecond time scale, while thermally activated switching occurs at much larger time scales [8]. The transition between these two switching regions lies between 1 and 10ns, which is depicted in Figure 7.4. The dynamics of precessional switching are well described by the Landau-

Lifshitz-Gilbert equation (LLGE) [98, 99], given by:

$$\frac{\partial \vec{m}}{\partial t} = -\gamma M_S \vec{m} \times \left(\vec{h}_{eff} - \alpha \frac{\partial \vec{m}}{\partial t} \right). \tag{7.2}$$

Equation 7.2, with the addition of Slonczewski's spin-torque transfer term [9], will be discussed in more detail in later sections.

Switching occurs on much longer time scale when the current though the MTJ is less than the critical switching current [96]. In the thermally activated regime, the switching current is a function of pulse duration τ:

$$I_C = I_{C0} \left[1 - \frac{\ln(\tau/\tau_0)}{\Delta} \right], \tag{7.3}$$

where Δ is the thermal stability of the MTJ, τ_0 is the natural time constant, and I_{C0} is the critical switching current [100].

7.2.2.3 Probabilistic Switching

Due to thermal agitation, the initial angle between the magnetizations of the fixed and free layers are in constant flux [102]. Combined with other finite temperature effects, this leads to a time-varying critical switching current [103]. This effect is very well modeled as a single critical switching current with a probabilistic distribution [104]. Figure 7.5 shows one such measurement of the probabilistic distribution for a $135 \times 65 nm^2$ CoFeB/MgO/CoFeB device. For memory applications, devices exhibiting very sharp transitions are highly desirable [105].

7.2.3 Tunnel Magnetoresistance Temperature Dependency

The sensitivity of TMR to temperature is well documented in literature [84,106,107,108,109]. The effect at zero bias voltage is very well described by the Jullière conductance model [106]. The Jullière model decomposes the conductance of the MTJ into two parts: (i)

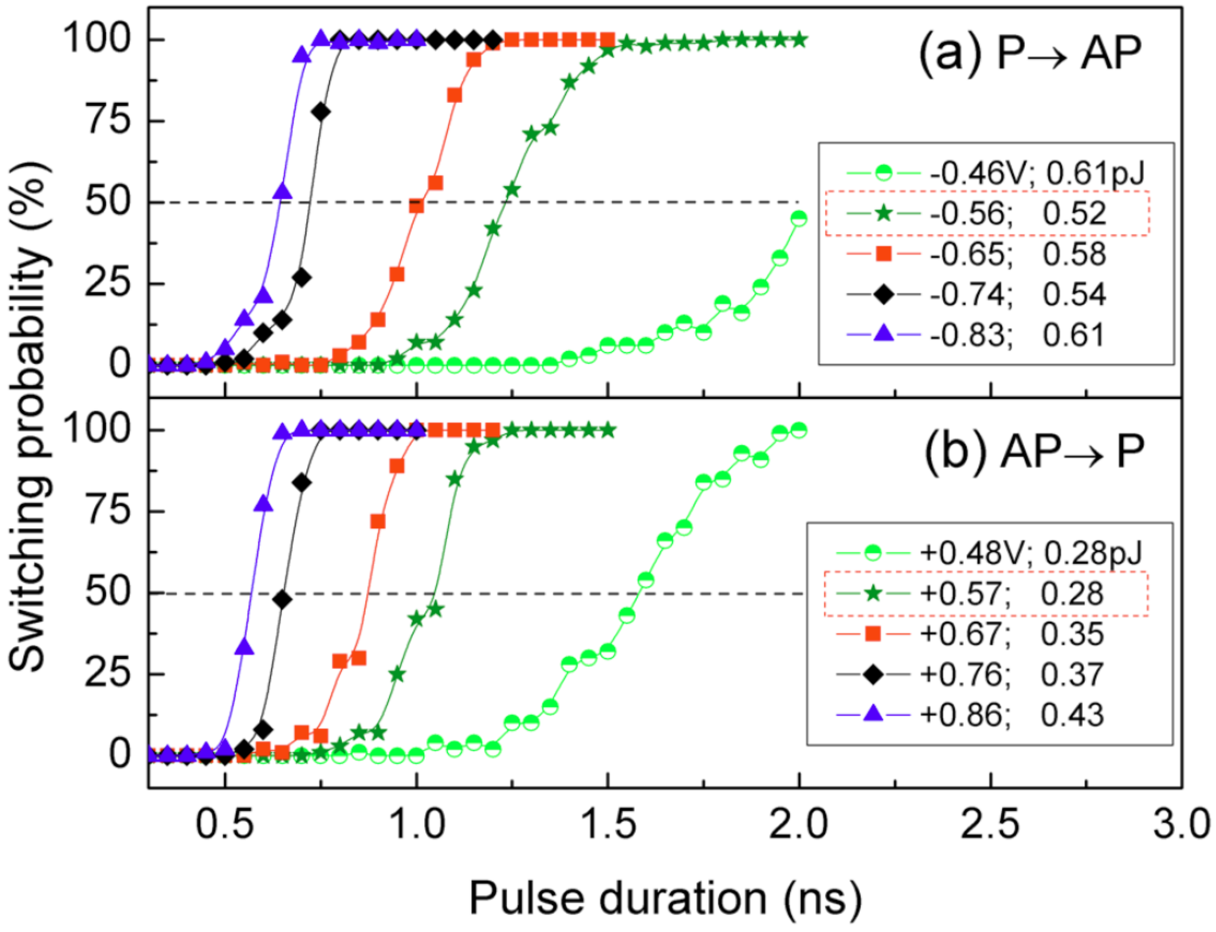


Figure 7.5: Switching probability vs. pulse duration for (a) $P \rightarrow AP$ and (b) $AP \rightarrow P$. Reproduced with permission from Zeng et al. [101].

G_T, the conductance due to direct elastic tunneling, and (ii) G_{SI}, the conductance due to imperfections in the insulating layer (assumed to be unpolarized). The total conductance (G), as a function of the angle θ is given by:

$$G(\theta) = G_T \{1 + P_1 P_2 \cos(\theta)\} + G_{SI}, \tag{7.4}$$

where P_1 and P_2 are the factors of spin-polarization for the two ferromagnetic layers, and $\theta = 0°$ for parallel and $\theta = 180°$ for anti-parallel magnetization. The temperature dependence of spin-polarazation has been extensively studied and shown to be:

$$P\left(T\right) = P_0\left(1 - \alpha_{sp}T^{3/2}\right). \tag{7.5}$$

It should be noted that variations in G_T due to temperature are almost negligible, whereas $G_{SI} \propto T^{4/3}$ has been confirmed both theoretically and experimentally [110].

7.2.4 Bias Voltage Effects

The Jullière conductance model is not perfect, being only able predict TMR at zero bias voltage [107]. Figure 7.3 illustrates the effect of the so called "zero bias anomaly" in an MTJ structure [111]. The source of the bias voltage dependence of TMR is still not very well understood [112]. However, it is suspected that elastic currents play a role at low voltages [113] and redistribution of the density of states at higher voltages [112]. At higher voltages, Simmons' formula can be used to model the density of states to predict degradation of TMR to a bias voltage [108].

7.2.5 Other Important MTJ Characteristics

7.2.5.1 Self Induced Heating

Due to small device sizes and large write currents, the power density of a write operation in an MTJ can be very high. These high power densities can lead to localized heating or self induced heating in MTJs [114]. Hotspots (weak areas in the insulating barrier) and pinholes (direct contact between the magnetic layers) cause nonuniform current flow through the MTJ [115]. This leads to nonuniform heating across the tunneling barrier, affecting spin-polarization efficiency and causing inelastic electron scattering [115]. Simulations show that consecutive write opperations produce a 9-15°C increase in the temperature of the MTJ [114]. Additionally, a large number of writes followed by a read leads to degraded sensing margin. Self induced heating is exploited as the writing mechanism in Thermal Assisted Switching

MRAMs (TAS-MRAMs) [116]. However, in STT-MRAMs, lower RAs are generally used to avoid self induced heating [114].

7.2.5.2 Backhopping

Backhopping is a recently discovered phenomenon, whereby increasing the bias voltage beyond the apparent switching threshold causes the MTJ to precess back and forth before settling to its original state [117]. This results in a lowered probability of switching at bias voltages beyond the threshold, causing non-monotonicity in the probability switching curves [118]. Backhopping is also much more pronounced in switching from an antiparallel to a parallel state [117, 118]. This suggests that backhopping is related to the interlayer exchange coupling between the free and fixed layers. Backhopping is more pronounced on longer time scales, where self induced heating could be lowering the thermal energy barrier and causing hot-electron events [117]. Another explanation is that certain noise processes (discussed in the next section) might be responsible [118].

7.2.5.3 Noise

Many different mechanisms are responsible for noise in MTJs. Among these are thermal noise (Johnson-Nyquist), shot noise (current), flicker noise ($1/f$), random telegraph noise (RTN), and noise due to charge-trapping in the oxide barrier [105, 119, 120, 121, 122]. Due to the strong coupling between magnetization and junction resistance in MTJs, noise in the magnetic domain is responsible for random resistance fluctuations [119]. These resistance fluctuations are responsible for $1/f$ noise as well as RTN [122]. Magnetic impurities inside the tunneling barrier are responsible for charge-trapping [119].

Thermal noise dominates at low bias voltages before quickly being overpowered by shot noise [121]. At room temperatures, shot noise typically dominates for bias voltages greater than $50mV$ [120]. The thermal noise of an MTJ is given by $S_V = 4k_BTR_{MTJ}$, where k_B is Boltzmann's constant, T is in Kelvin, and R_{MTJ} is the resistance of the MTJ [105]. Similarly, shot noise can be expressed as $S_V = 2eIR_{MTJ}^2$, where e is the charge of an electron and I is the current through the device [105].

Another significant contribution to low-frequency noise is due to domain wall hopping between pinning sites [105, 119]. These pinning sites are created by edge roughness, interface defects, bulk defects, and random film anisotropy [119]. The low-frequency noise characteristics of an MTJ can be significantly reduced by improving the smoothness of the ferromagnetic/insulator interface [122].

7.3 Spintronic Memories

7.3.1 Field-Induced Magnetic Switching

Conventional, first-generation MRAMs use field-induced magnetic switching (FIMS) to toggle the MTJ between its parallel and antiparallel states [123]. Figure 7.6 shows the layout and cross-sectional view of a FIMS-MRAM, with the word lines and bitlines organized into a crosspoint architecture. To write a cell, a synchronized pulse of current is applied to the desired word and bit lines, creating a strong magnetic field at the intersection of the two wires and causing the MTJ to switch to the desired state [6]. A small access transistor is also required to read the resistive state of each MRAM cell [124].

Like any crosspoint architecture, conventional MRAMs suffer from a potentially serious write disturbance problem, or the "half-select" problem [6]. In the half-select problem, a small amount of noise on the word line (bit line) of any cell whose bit line (word line) is selected, but whose word line (bit line) is not, might cause accidental writing. Additionally, as technologies are scaled down, more current is required to write an FIMS-MRAM cell [4].

7.3.2 TAS-MRAM

Thermally assisted switching MRAM (TAS-MRAM) exploits the temperature dependence of ferromagnetic materials to address the shortcomings of FIMS-MRAMs. Structurally, TAS-

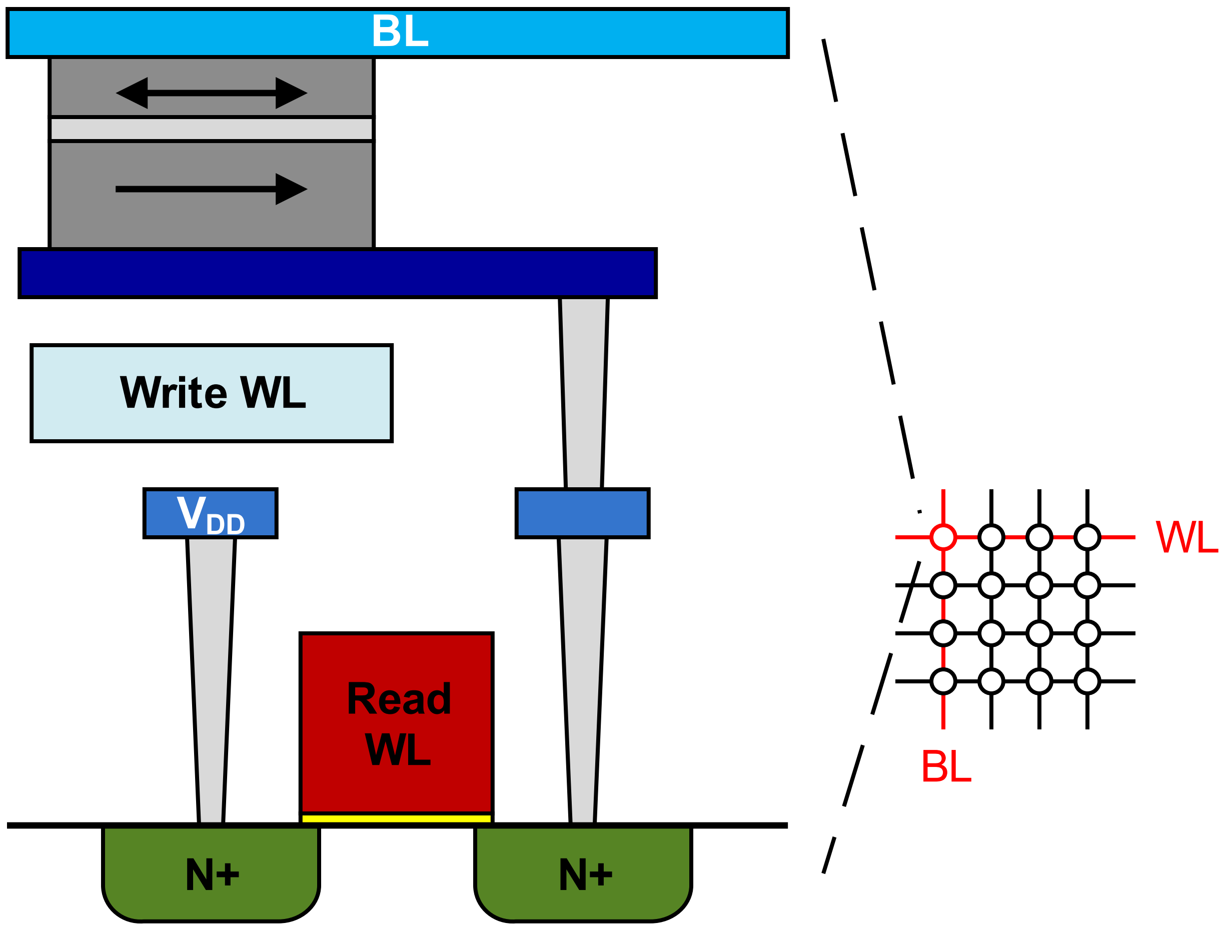


Figure 7.6: The cross-sectional view of a FIMS-MRAM in a crosspoint architecture.

MRAM is identical to FIMS-MRAM, with the addition of a heating element or layer next to the free layer of the MTJ. During the write operation, current is driven through the MTJ by the read access transistor, causing joule heating of the free layer [125]. Heating of the MTJ serves two purposes. First, switching of the MTJ free layer becomes easier to accomplish at higher temperatures, reducing the overall writing current [126]. Second, since switching now requires heating of the desired memory cell, the half-select problem is gone [127]. The free layer of an MTJ for TAS-MRAM is also typically engineered to have a low Currie temperature for enhanced thermal switching, allowing the writing currents to remain constant and even scale down in more advanced technology nodes [126].

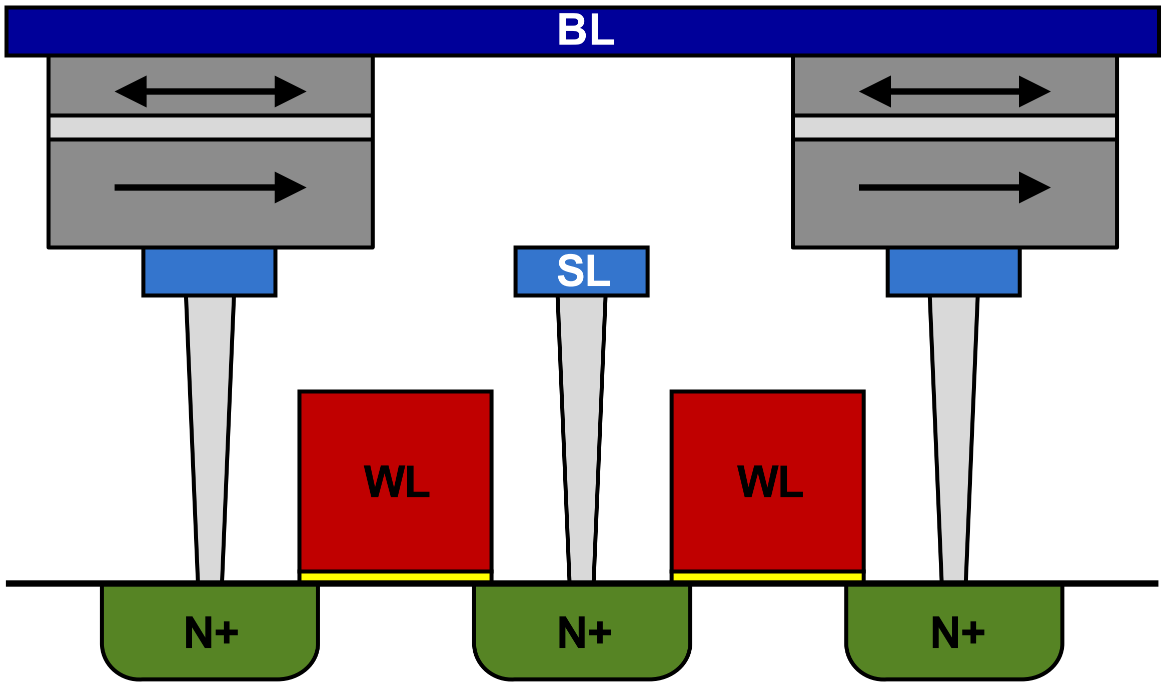


Figure 7.7: The cross-sectional view of two STT-MRAM cells with shared source lines.

7.3.3 STT-MRAM

Spin-torque transfer MRAM (STT-MRAM) operates on an entirely different principle from FIMS-MRAM and TAS-MRAM. Rather than using an indirect current to generate a magnetic field, spin-torque transfer (STT) based switching uses a spin-polarized current through the MTJ to accomplish device switching [9]. In STT-based switching, toggling of the MTJ is roughly determined by the current density [10]. As the area of the MTJ device decreases, so does the writing current, enabling much better scaling than either FIMS-MRAM or TAS-MRAM. Architecturally, STT-MRAMs are also much simpler than conventional MRAMs [11]. Figure 7.7 shows the layout of two STT-MRAM cells utilizing a shared source line for improved density.

7.3.4 MeRAM

It has been observed that the interface between an MgO tunneling oxide and a CoFeB ferromagnetic layer can exhibit strong perpendicular magnetic anisotropy (PMA) [66, 128,

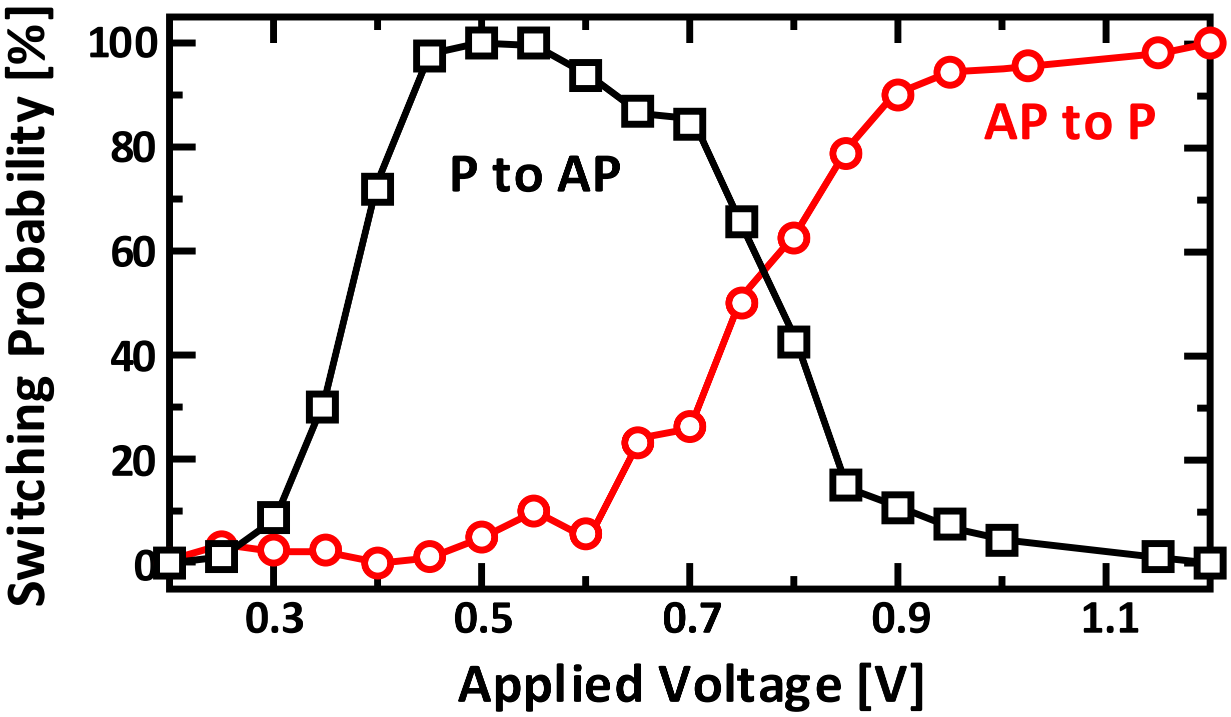


Figure 7.8: Measured probability of switching curves for the VCMA-based MTJs. Data was obtained using 100 repetitions for each voltage with 100ms pulse widths. The combination of VCMA and STT effects allow for a unipolar set/reset switching scheme with switching voltages 0.5V and 1.1V respectively. Measurements on similar devices with a thicker MgO barrier (~20 times larger resistance) showed only the first (VCMA-induced) switching described in this work, with the second (STT-induced) switching absent at larger voltages.

129]. Furthermore, it has been discovered that this PMA is sensitive to a voltage applied across the MgO-CoFeB junction [130, 131, 132]. Using a voltage to modulate the PMA of an MTJ is called the voltage controlled magnetic anisotropy (VCMA) effect and it is the basic principle of operation behind Magnetoelectric (i.e. electric-field-controlled) Random Access Memory (MeRAM). Fundamentally, no current flow is required for the VCMA effect. As such, MeRAM has the potential to be significantly more energy and area efficient compared to STT-MRAM. However, VCMA is often used in conjunction with to traditional STT switching [130, 132, 133, 134, 135]. The use of VCMA and STT allows the MTJ to switch in both directions (write opposite bits of information) using voltages of the same polarity, but with different magnitudes. This enables a diode-MTJ crossbar architecture, with the potential for a sub-1F^2 effective cell size, significantly denser than traditional (purely current-switched) STT-MRAM. The probability of writing as a function of applied voltage for the MTJs used in such a crossbar array is shown in Figure 7.8, demonstrating their unipolar set/reset characteristics.

7.4 Modeling MTJ Characteristics

Recent advances in MgO-based MTJs show strong potential for STT-MRAMs [136]. STT-MRAM and MeRAM have the potential to rival the densities of DRAM, the speed of SRAM, and is non-volatile without degrading over time like Flash [5]. The greatest hindrance in the design of spintronics circuits is the lack of a compact MTJ model capable of accurately modeling temperature and voltage dependencies. Capturing these dependencies, in a compact model compatible with circuit simulators, is crucial for performing accurate Monte Carlo simulations to place yield and performance bounds on STT-MRAM and MeRAM. This section presents such a model implemented in Verilog-A. The model's simulation results were also compared to a model implemented using the LLG Micromagnetics Simulator [137] and actual device measurements from 135nm by 65nm CoFeB/MgO/CoFeB MTJs.

7.4.1 Magnetization Dynamics

The precessional motion of magnetization ($\vec{M}$) of the free layer of a MTJ, in the presence of an external magnetic field ($\vec{H}_{eff}$), can be very accurately modeled by the LLGE, Equation 7.2 [100]. With the introduction of Slonczewski's spin-torque transfer term [9], the normalized LLGE with STT is given by:

$$\frac{\partial \vec{m}}{\partial t} = -\gamma M_S \vec{m} \times \left(\vec{h}_{eff} + \frac{J_e}{J_C} b\left(\theta\right) \left(\vec{m} \times \vec{p}\right) - \alpha \frac{\partial \vec{m}}{\partial t} \right), \tag{7.6}$$

where $M_S = |\vec{M}|$, γ is the absolute value of the gyromagnetic ratio ($\gamma_e \mu_0$), $\vec{m}$ is the unit vector in the direction of $\vec{M}$, $\vec{p}$ is the unit vector in the direction of the magnetization of the

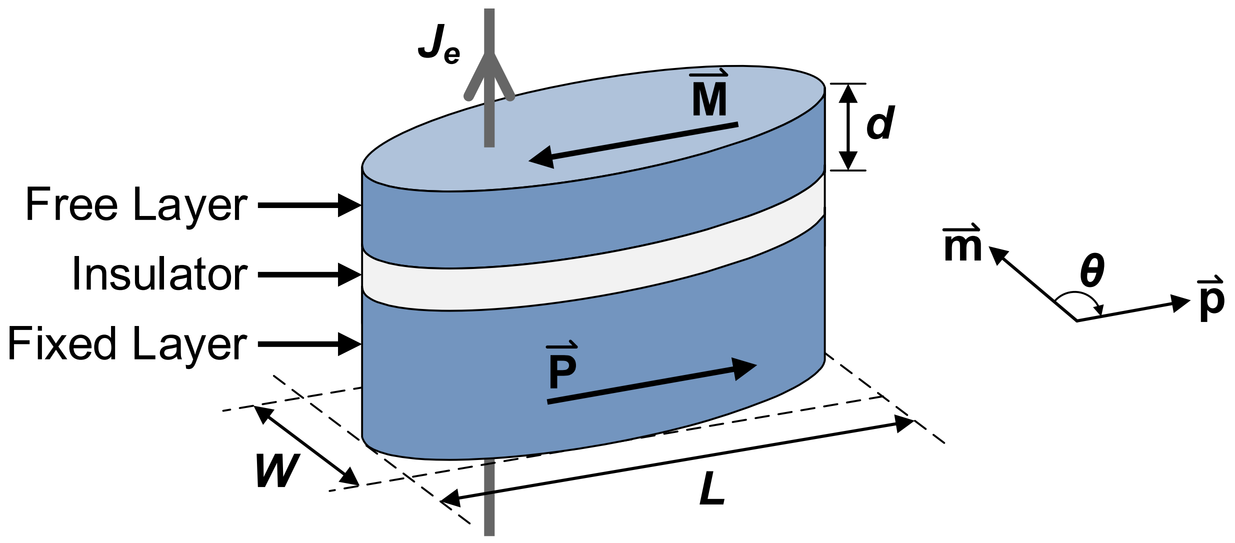


Figure 7.9: Sketch of basic MTJ structure.

fixed layer, $\vec{h}_{eff} = \vec{H}_{eff}/M_S$, J_e is the current density (see Figure 7.9), θ is the angle between $\vec{m}$ and $\vec{p}$, and $\alpha > 0$ is the material-dependent Gilbert damping constant. The efficiency factor of spin-polarization ($b(\theta)$, see Figure 7.10) is defined as:

$$b(\theta) = \left[-4 + (1+P)^3 \frac{\{3+\cos(\theta)\}}{4P^{3/2}}\right]^{-1}, \tag{7.7}$$

where P is the percentage of electrons polarized in the $\vec{p}$ direction. The switching current density (J_C) has been modified to include thermally-activated switching [100]. For a constant pulse of duration τ, J_C is given by:

$$J_C = J_{C0}\left[1 - \frac{\ln(\tau/\tau_0)}{\Delta}\right], \tag{7.8}$$

where Δ is the thermal stability of the MTJ and $\tau_0 = (\gamma M_S)^{-1}$ is the natural time constant. Furthermore, the characteristic current density (J_{C0}) is defined as:

$$J_{C0} = \gamma M_S \frac{e M_S d}{g_e \mu_B}, \tag{7.9}$$

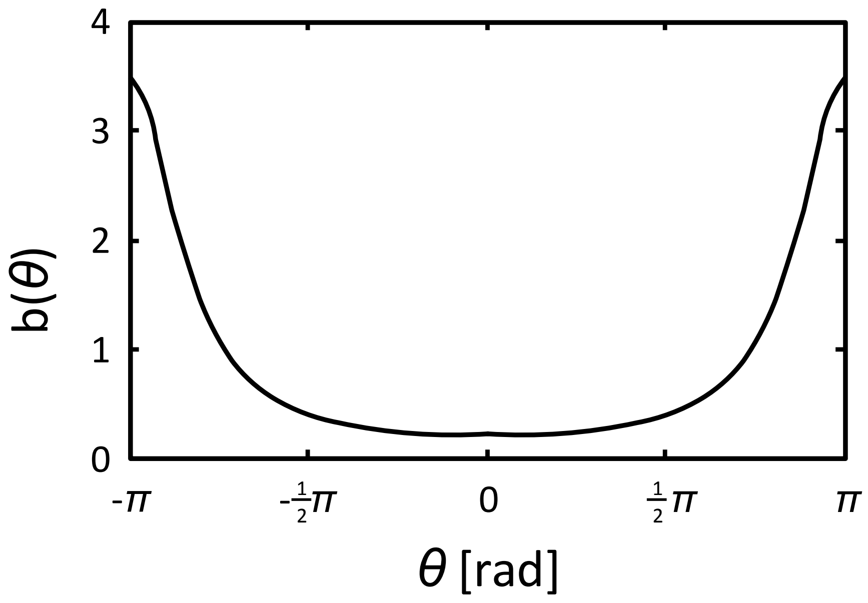


Figure 7.10: Magnitude of the efficiency factor of spin-polarization vs. θ for $P = 0.65$.

where e is the absolute value of electron charge, d is the thickness of the free layer, g_e is the Landé factor for electrons, and μ_B is the Bohr magneton [138].

7.4.2 Effective Magnetic Field

The effective magnetic field ($\vec{H}_{eff}$) is given by:

$$\vec{H}_{eff} = \vec{H}_{ext} + \vec{H}_{dem} + \vec{H}_{an} - \vec{H}_{VCMA} + \vec{H}_{th}, \tag{7.10}$$

where $\vec{H}_{ext}$ is the external applied magnetic field, $\vec{H}_{dem}$ is the demagnetization field, $\vec{H}_{an}$ is the magnetocrystalline anisotropy field, $\vec{H}_{VCMA}$ is the Voltage Controlled Magnetic Anisotropy (VCMA) field, and $\vec{H}_{th}$ is effective contribution of thermal noise.

7.4.2.1 External and Demagnetization Fields

The demagnetization field (shape anisotropy) varies with the geometry of the free layer and is modeled as $\vec{H}_{dem} = N\vec{M}$. If the free layer is assumed to be a very flat ellipsoid, the factors of the demagnetization tensor N, calculated by Osborn [139], are:

$$N_X = \frac{d}{L}\left(1-e^2\right)^{1/2}\frac{K-E}{e^2}, \tag{7.11}$$

$$N_Y = \frac{d}{L}\frac{K-\left(1-e^2\right)E}{e^2\left(1-e^2\right)^{1/2}}, \tag{7.12}$$

$$N_Z = 1-\frac{d}{L}\frac{E}{\left(1-e^2\right)^{1/2}}, \tag{7.13}$$

where K and E are the complete elliptic integrals of the first and second kind whose argument is:

$$e = \left(1-W^2/L^2\right)^{1/2}. \tag{7.14}$$

7.4.2.2 Voltage Controlled Magnetic Anisotropy

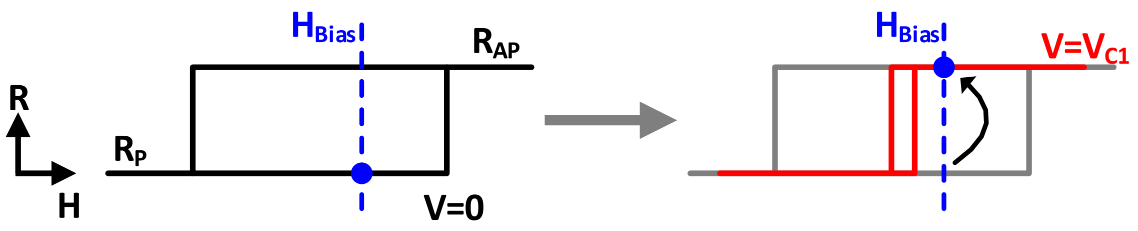
The intrinsic PMA of the MTJ ($\vec{H}_{an}$) and the VCMA effect ($\vec{H}_{VCMA}$) are the third and fourth components of $\vec{H}_{eff}$ in the LLG equation.

$$\vec{H}_{an} = \frac{2K_i}{d\mu_0 M_s}m_z\hat{z} \tag{7.15}$$

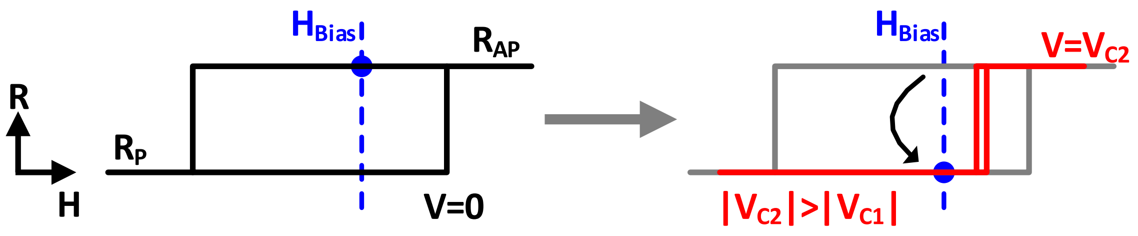
$$\vec{H}_{VCMA} = \frac{2\xi V}{\mu_0 M_s t_{ox} d}m_z\hat{z} \tag{7.16}$$

K_i is the PMA constant, ξ is the VCMA constant, V is the voltage across the MTJ, and t_{ox} is the thickness of the oxide layer. Notice that both terms are in the z-direction and the strength of each is proportional to m_z, and the VCMA term subtracts from the PMA term in $\vec{H}_{eff}$. This means that applying a positive (negative) voltage across the MTJ reduces (increases) its PMA, and thus reduces (increases) its coercivity. This is consistent with experimental results from [133]. A VCMA constant as high as 37 $fJ/(V\dot{m})$ has been experimentally observed [130].

Figure 7.11 shows schematically how voltage pulses of the same polarity, but different



(a) Switching P to AP.



(b) Switching AP to P.

Figure 7.11: Writing of the VCMA-based MTJs in the P to AP direction is accomplished by a voltage V_{C1}, which reduces the coercivity of the free layer and results in a single available state at the bias field H_{Bias} provided by the fixed layer. Further increasing the magnitude of the voltage to V_{C2} allows STT to switch the device in the opposite (e.g., AP to P) direction [133, 140].

amplitudes, can be used to switch a MeRAM device in opposite directions [133]. The free layer of the MTJ devices is subject to a bias field H_{Bias}, due to the combination of the stray field from the pinned layer, as well as an externally applied magnetic field (which is fixed throughout the experiment, and can be removed in an optimized device by proper design of the fixed layer to apply the required H_{Bias}). This field favors one of the states in the free layer (e.g., AP state in Figure 7.11), but is small enough not to compromise the bistability of the bit. This is shown schematically in the resistance R versus magnetic field H curves at equilibrium ($V = 0$) in Figure 7.11, where the magnetic field H is along the easy axis of the free layer. Once a voltage V_{C1} of the proper polarity is applied, the modification of the interfacial perpendicular anisotropy via the VCMA effect [132] reduces the coercivity of the free layer, resulting in a single available state at H_{Bias} and, thus, P→AP switching in Figure 7.11(a). When the voltage is increased to V_{C2}, the coercivity will further decrease, but current-induced effects also become increasingly important. If the polarity is designed such that STT favors the opposite free layer direction compared to the bias field H_{Bias}, a voltage pulse V_{C2} will thus induce AP→P switching in Figure 7.11(b). This allows for a unipolar set/reset write scheme, where voltage pulses of the same polarity, but different amplitudes, can be used to switch the device between the P and AP states [133,140]. It should be noted that applying negative voltages across the MTJ strengthens the PMA and the stability of the devices present state and can be used to eliminate any potential read disturbance in MeRAM [141,142].

7.4.2.3 Thermal Noise

The last and final component of $\vec{H}_{eff}$ in the LLG equation is the thermal noise term $\vec{H}_{th}$. Thermal noise creates random fluctuations in the free layer magnetization and, therefore, in the MTJ resistance.

$$\vec{H}_{eff} = \vec{\sigma}\sqrt{\frac{2k_B T\alpha}{\mu_0 M_s \gamma V \delta t}}, \tag{7.17}$$

where k_B is Boltzmann's constant, T is the absolute temperature in Kelvin, V is the volume of the free layer, and δt is the simulation time step. $\vec{\sigma}$ is a unit vector whose x, y, and z components are independent Gaussian random variables with $\mu = 0$ and $\sigma = 1$. These components are produced using Verilog-A built-in random number generator functions. One of the major consequences of thermal noise is that the switching behavior becomes probabilistic [24-25]. This aspect of the model is essential for measuring switching probabilities and for simulating any switching behavior in the thermally activated regime.

7.4.2.4 Temperature Dependencies

In the dynamic equations, only the magnetization saturation (M_S) and the spin-polarization (P) vary with temperature. For temperatures below the Curie temperature (T_C), we can use the Weiss theory of ferromagnetism [143] to model:

$$M_S(T) = M_{S0}(1 - T/T_C)^{\beta}, \tag{7.18}$$

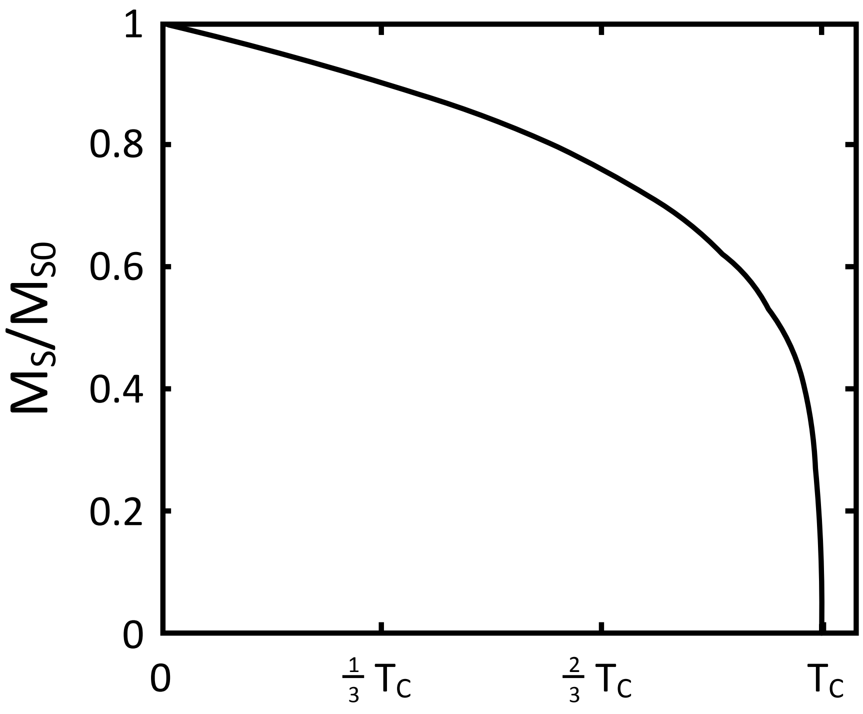


Figure 7.12: Normalized plot of magnetization saturation for generic ferromagnetic materials.

where M_{S0} is the magnetization saturation at absolute zero and β is the material-dependent critical exponent (see Figure 7.12) [144]. Similarly, the temperature dependence of spin-polarization has been extensively studied and shown to be:

$$P(T) = P_0 \left(1 - \alpha_{sp} T^{3/2}\right) \tag{7.19}$$

where P_0 is the spin-polarazation at absolute zero and α_{sp} is a material and geometric dependent constant [110].

7.4.3 Tunnel Magnetoresistance

Temperature variations in MTJ conductance $(G(\theta))$ are modeled in Shang et al. [110] by modifying the Jullière model. Jullière's model, Equation 7.4, is reproduced here with $P_1 =$

$P_2 = P$:

$$G(\theta) = G_T \left\{1 + P^2 \cos(\theta)\right\} + G_{SI}. \tag{7.20}$$

As a reminder, the variation of G_T due to temperature is negligible, whereas $G_{SI} \propto T^{4/3}$. Using $\theta = 0°$ for parallel magnetization and $\theta = 180°$ for anti-parallel, the tunnel magnetoresistance with zero applied bias voltage (TMR_0) can be expressed as:

$$TMR_0(T) = \frac{2P_0^2\left(1 - \alpha_{sp}T^{3/2}\right)^2}{1 - P_0^2(1 - \alpha_{sp}T^{3/2})^2 + \frac{G_{SI}(T)}{G_T(T)}}. \tag{7.21}$$

The Jullière model fails to predict the effects of a bias voltage on TMR [108]. However, this can be rectified with the addition of a simple fitting function:

$$TMR(T, V) = \frac{TMR_0(T)}{1 + \left(\frac{V}{V_0}\right)^2}, \tag{7.22}$$

where V_0 is the voltage at which TMR is halved.

7.4.4 Heun's Method

P. Horley et al. performed an investigation of numerical simulation techniques for solving the LLG equation [26]. They concluded that a second order approach is required to obtain a correct solution and that Heun's method is a reasonable compromise between accuracy and computation time. Given an ordinary differential equation of the form $\dot{y} = f(t, y(t))$ with $y(t_0 = y_0)$, the Fundamental Theorem of Calculus tells us the following:

$$y(t_{i+1}) = y(t) + \int_{t_i}^{t_{1+1}} \dot{y}(u)du. \tag{7.23}$$

The simplest way of numerically approximating the solution is Eulers method:

$$y_{i+1} = y_i + hf(t_i, y_i). \tag{7.24}$$

where h is the computation time step. This amounts to approximating the integral as a Riemann sum, in other words it estimates the area under the function $\dot{y}(t)$ with a series of rectangles. Heuns method is more accurate because it approximates the integral using the Trapezoidal Rule:

$$\tilde{y}_{i+1} = y_i + hf(t_i, y_i) \tag{7.25}$$

and

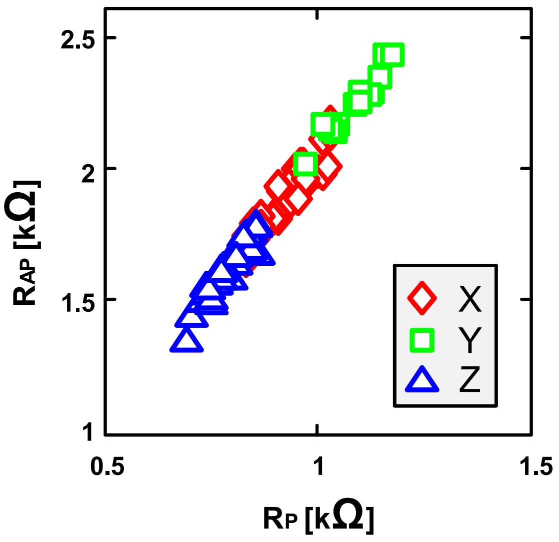
$$y_{i+1} = y_i + \frac{h}{2}[f(t_i, y_i) + f(t_{i+1}, \tilde{y}_{i+1})]. \tag{7.26}$$

The real benefit here is that the accuracy of Heun's method increases quadratically with a decrease in the time step whereas the accuracy of Euler's method only increases linearly [27]. For all these reasons, the compact MTJ model uses Heun's method to solve the LLG equation.

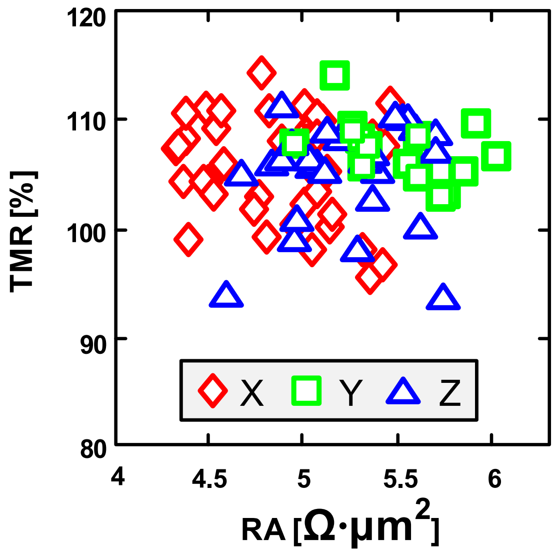
7.4.5 Statistical Characterization of MTJ Devices

7.4.5.1 MTJ Device Variability

While statistical variation of CMOS is generally well understood, similar characteristics for MTJs have not been well documented. This section uses a combination of fundamental equations and measured device characteristics to model the statistical behavior of MTJs. Figure 7.13(a) contains a plot of measured R_P vs. R_{AP} for 105 MTJ nanopillars of varying size and target RAs. Variations in resistance and TMR are due to a combination of lithographic variations in the physical dimensions of the nanopillar and minute variations in the thicknesses of the up to 20 different layers in state-of-the-art MTJ processes [145]. The cumulative effects of these variations on RA and TMR can be easily measured [146], as



(a) Switching P to AP.



(b) Switching AP to P.

Figure 7.13: Measured (a) R_{AP} vs. R_P and (b) TMR vs. RA for MTJ nanopillars measuring $150 \times 45nm^2$ (X), $130 \times 50nm^2$ (Y), and $170 \times 45nm^2$ (Z)

	X	Y	Z
TMR [%]	105.7	107.3	105.3
σ_{TMR} [%]	4.7	2.7	4.6
RA $[\Omega \cdot \mu m^2]$	4.88	5.51	5.22
σ_{RA} $[\Omega \cdot \mu m^2]$	0.342	0.297	0.311

Table 7.1: Measured device statistics.

shown in Figure 7.13(b) and Table 7.1.

7.4.5.2 Scaling of MTJ Current and Resistance

The resistance and switching current can be modeled using a precessional-based switching model, modified to include thermally-activated switching [100]. The switching current of an MTJ in the precessional region, for a constant pulse of duration τ, is given by:

$$I_C = I_{C0}\left[1 - \frac{\ln(\tau/\tau_0)}{\Delta}\right], \tag{7.27}$$

where τ_0 is the natural time constant and I_{C0} is the critical switching current. This critical switching current [138] is given by:

$$I_{C0} = \frac{\alpha 4\pi e}{\eta \hbar} M_S^2 V, \tag{7.28}$$

where α is the Gilbert damping constant, η is the factor of spin polarization, $\hbar$ is the reduced Planck's constant, e is the elemental charge of an electron, M_S is the magnetization saturation of the free layer, and V is the volume of the free layer.

For an MTJ with free layer dimensions $l > w >> d$ the thermal stability of an MTJ is approximately [139,147]:

$$\Delta = \frac{E}{k_B T} = \frac{H_K M_S}{2 k_B T} V \approx d \left(\frac{1}{w} - \frac{1}{l} \right) \frac{M_S^2}{k_B T} V, \tag{7.29}$$

where k_B is Boltzmann's constant, T is the absolute temperature in Kelvin, H_K is the out-of-plane uniaxial anisotropy, and E is the energy of anisotropy [148,149].

Dimensional scaling is performed to maintain a constant Δ in order to ensure the long-term non-volatility of the STT-MRAM. Therefore, dimensions l and w of the MTJ are scaled by a factor λ to manipulate I_{C0} and $R_{P/AP}$, then to keep Δ constant, d must scale by $\lambda^{-1/2}$. This results in $I_{C0} \propto lwd \rightarrow \lambda^{3/2}$ and $R_{P/AP} \propto l^{-1} w^{-1} \rightarrow \lambda^{-2}$.

7.5 Model Verification

Implemented in Verilog-A, the compact model is comprised of two electrical terminals, an externally applied field vector, the initial state of magnetization, the demagnetization factors, and 13 device-specific parameters: 4 geometric parameters, 8 material-dependent parameters, and 1 empirically-derived parameter. The model was fitted to a 135nm by 65nm CoFeB/MgO/CoFeB MTJ (see Figure 7.14). For validation of the model, we compare to detailed micromagnetic simulations, previously published data, and experimental results from

Geometric Parameters		**LLGE Damping**		**Conductance**	
W	65 [nm]	α	0.05	G_T	1.07 [mS]
L	135 [nm]	**Magnetization Saturation**		G_{SI}	0 [mS]
d	1.8 [nm]	M_{S0}	1100 [emu/cc]	**Demagnetization Tensor (Calculated)**	
t_{ox}	0.9 [nm]	T_C	1420 [K]		
Spin Polarization		β	0.4	N_X	0.0113
P_0	0.725	**TMR V_{BIAS} Fitting**		N_Y	0.0198
α_{sp}	2×10^{-5} [$K^{-3/2}$]	V_0	0.5 [V]	N_Z	0.9689

Figure 7.14: Fitted MTJ parameters.

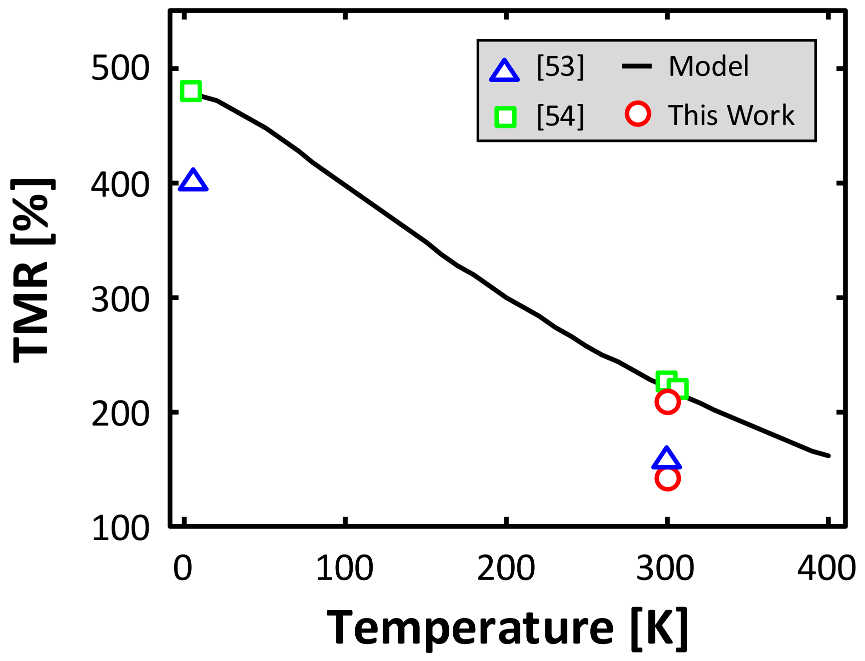


Figure 7.15: TMR vs. temperature: Verliog-A model (line), reported in [150] (triangles), reported in [151] (squares), and fabricated devices (circles).

fabricated MTJ nanopillars.

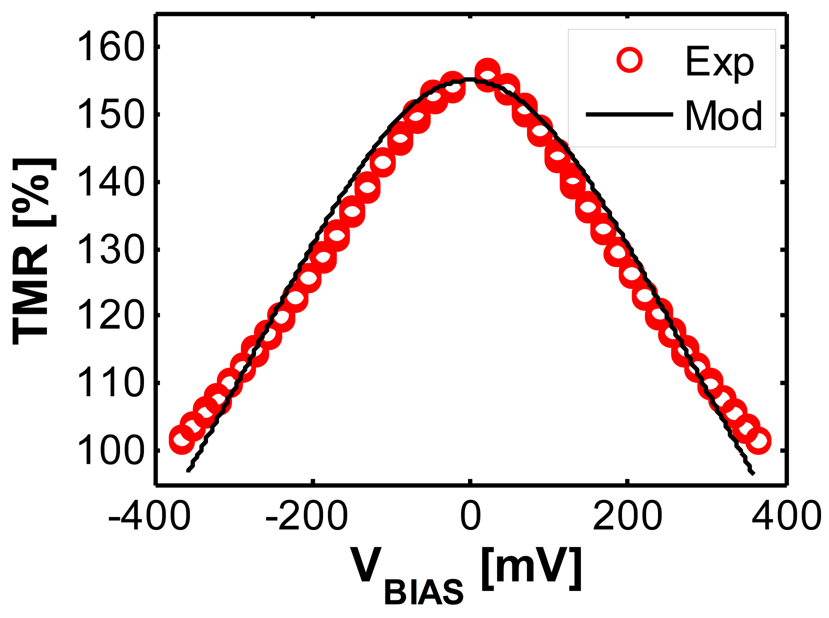


Figure 7.16: TMR vs. an applied bias voltage at 300K: Verliog-A model (black line) and fabricated devices (red circles).

7.5.1 Comparison to Measured Devices

The ability of Eqs. 7.20 and 7.21 to accurately model the temperature dependance of R_P and TMR is well established in literature. Shang et al. [110] managed to obtain excellent fitting for Al_2O_3 based MTJs before deviating at high temperatures due to the crystallization of the amorphous insulating layer. Similarly, Kou et al. [150] and Wiśniowski et al. [152] reported extremely good fitting for MgO- and IrMn-based devices respectively. Figure 7.15 contains a loosely fitted curve of (7.21) for limited empirical data, as well as reported TMR values from literature. An excellent fitting of (7.1) to experimental data was obtained (Figure 7.16), with an accuracy of $\pm 3\%$. The steady-state accuracy of (7.6) at modeling switching thresholds for an applied external field is quite good and can be seen in Figure 7.17.

7.5.2 Comparison to Micromagnetic Simulations

It is extremely difficult to accurately measure the switching characteristics of fabricated MTJs in the nanosecond regime. However, micromagnetic simulations are fully capable of

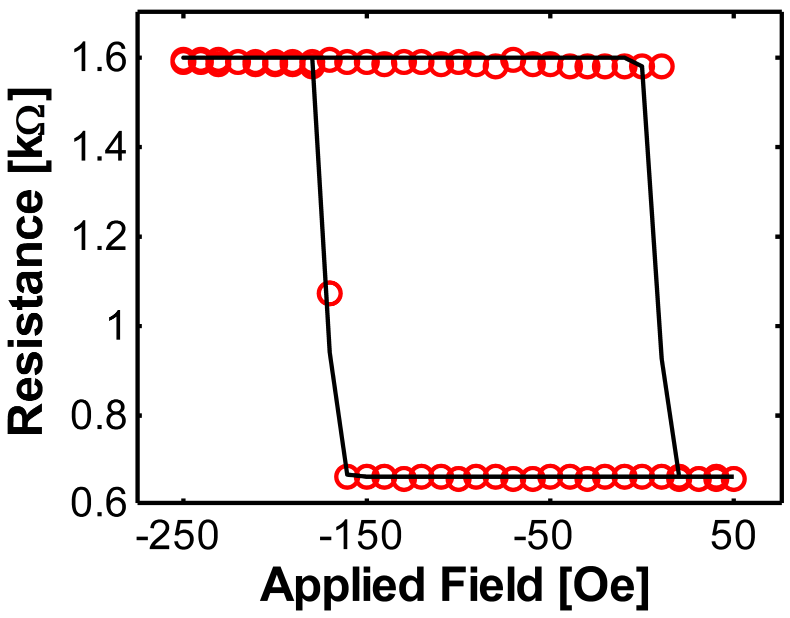


Figure 7.17: R-H hysteresis at 300K: Verliog-A model (black line) and fabricated devices (red circles).

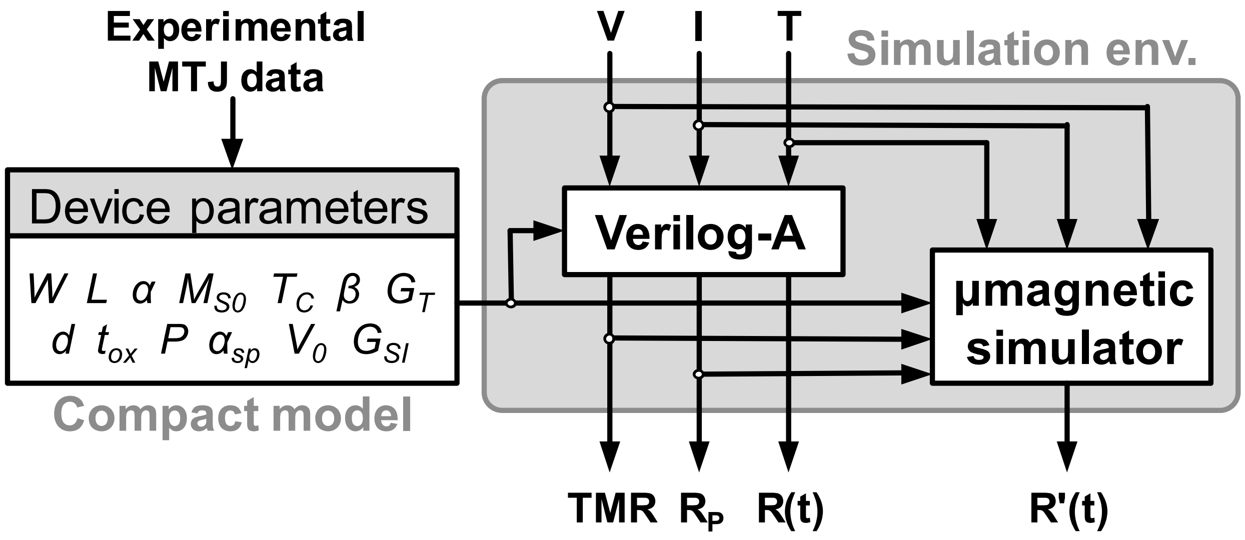


Figure 7.18: Process flow for evaluating Verilog-A model.

accurately predicting their behavior [137]. As such, micromagnetic simulations were used to evaluate the switching behavior of the Verilog-A model in the nanosecond regime at different temperatures (see Figure 7.18). Figure 7.19 shows the time evolution of the resistance model ($R(t)$) and the micromagnetic derived resistance ($R'(t)$) at 300K and 380K (expected

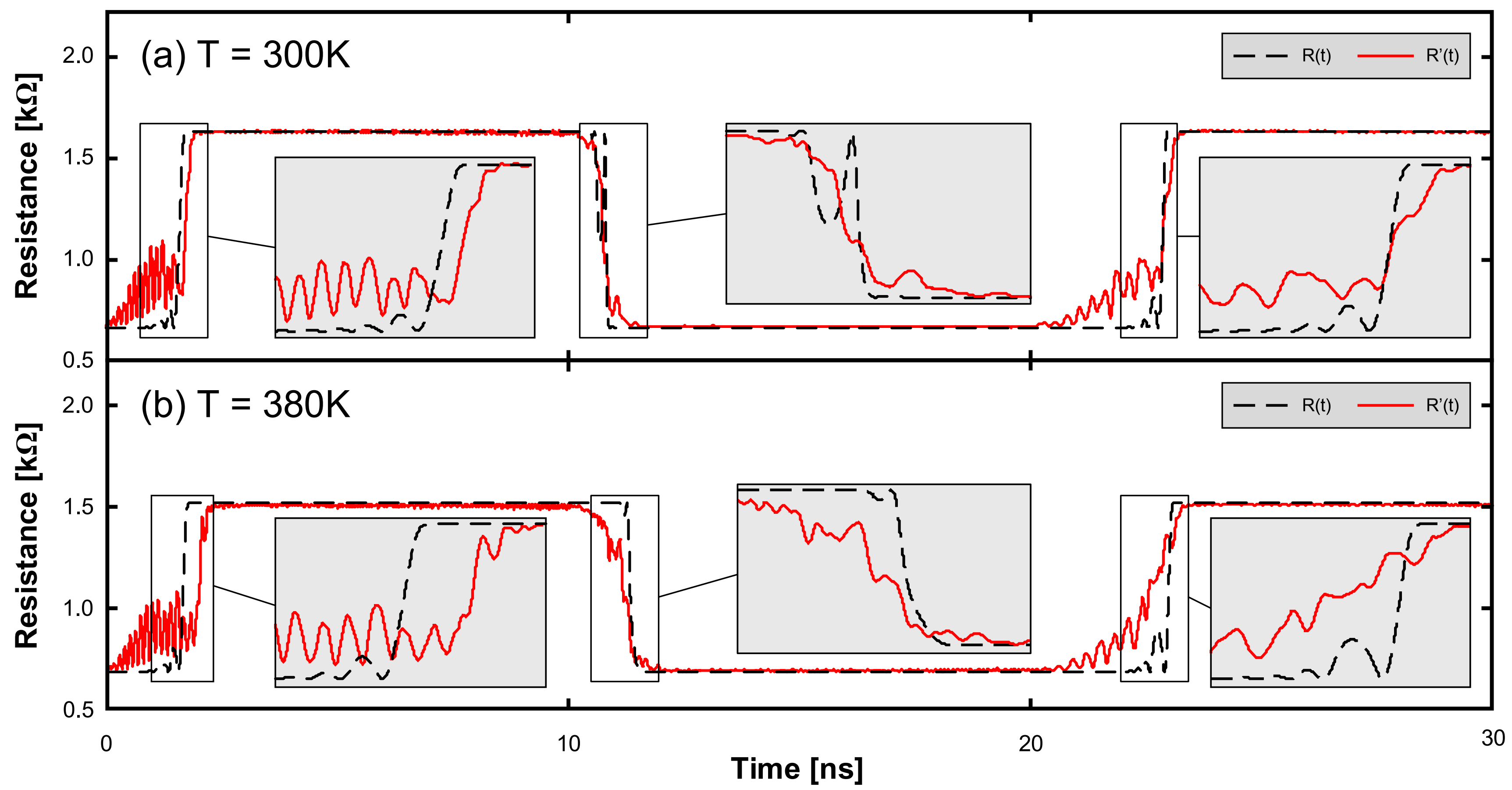


Figure 7.19: Resistance vs. time for an applied ±2V 100MHz square-wave at (a) 300K and (b) 380K. Verliog-A model ($R(t)$) is the black, dashed line and micromagnetic simulations ($R'(t)$) in the red, solid line.

operating temperature when integrated with CMOS).

Being based on a macrospin model, $R(t)$ does not account for non-uniformities in the free layer magnetization during switching. Despite this, the pre-switching oscillations and underdamped behavior of $R'(t)$ are still observed to a point in $R(t)$. This effect is captured by the shape anisotropy modeled by the demagnetization tensor (Eqs. 7.11, 7.12, and 7.13). Also, $R(t)$ manages to track the switching delay of $R'(t)$ across a wide range of temperatures and pulse shapes.

CHAPTER 8

STT-MRAM and MeRAM

The focus of this chapter is to introduce the discuss the details of the design and fabrication of several STT-MRAM and MeRAM chips. MTJ-CMOS device integration is also discussed along with a variety of different memory architectures.

8.1 MTJ/CMOS Integration

As mentioned before, MTJs are well suited for integration into a commercial CMOS process flow. In this flow, the deposition of the insulating oxide barrier is critical to the performance of the MTJ. If the layer is too thin ($< 0.7nm$) the MTJ does not exhibit any TMR, due to the formation of pin holes and soft points shorting the barrier. If the layer is too thick ($> 2.5nm$), then the resistance of the device is too large [153]. The deposition surface also needs to be very smooth, whereas typical Al interconnects (with a $\langle 111 \rangle$ texture) are far too rough. However, the Cu interconnects available in the thin metal layers of modern state-of-the-art fabrication process are ideal for MTJ deposition [154]. MTJs are typically integrated after the thin Cu layer, usually M4 in most processes. Figure 8.1 shows the side view of a typical 1T-1MTJ with full integration at M4. MTJ pillar dimensions down to 30nm can be accomplished with e-beam lithography, focused ion beam etching, or double patterning [155].

8.2 STT-MRAM Memory Architectures

Several different types of memory architectures exist for STT-MRAMs. At the cell level, many architectures are tailored to certain MTJ characteristics, more specifically to the ratio of the critical writing currents $I_C(P \rightarrow AP)$ and $I_C(AP \rightarrow P)$. Other cell architectures attempt to exploit the different thresholds between reading and writing current to increase effective memory density. At the array level, several different subarraying techniques are employed to maximize performance and minimize area.

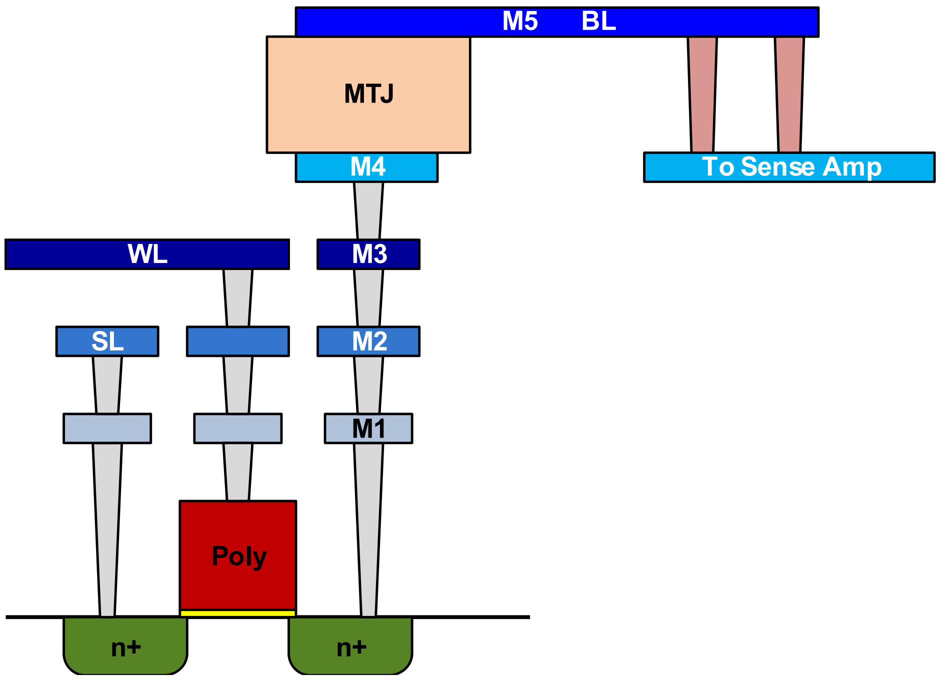


Figure 8.1: MTJ/CMOS integration at M4.

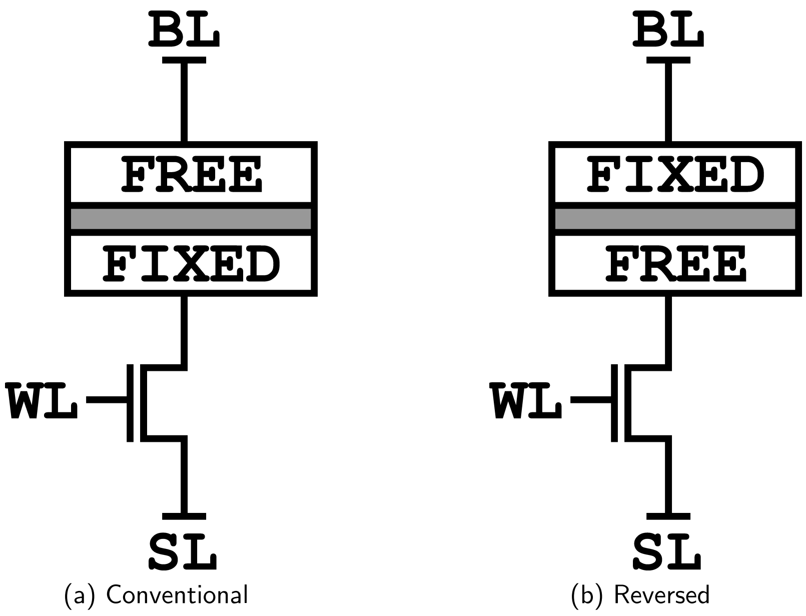


Figure 8.2: 1T-1MTJ memory cell architectures.

8.2.1 Cell Architectures

8.2.1.1 1T-1MTJ

There are two widely used 1T-1MTJ STT-MRAM cell architectures, the "conventional" cell (Figure 8.2(a)) and the "reverse" cell (Figure 8.2(b)) [156]. The "conventional" architecture gets its name from that fact that most MTJ are deposited with the fixed layer on the bottom. A smooth deposition surface is required to form a high quality pinning layer capable of generating the fixed layer [157]. The surface roughness introduced by various film deposition steps generally makes depositing the pinning layer on the top of the MTJ stack impractical. This means that it is easier to connect the fixed layer of the MTJ to the access transistor and the free layer to the bitline.

The "reverse" structure is built exactly as it sounds, with the fixed and free layers connected in the reverse fashion of the "conventional" cell. The "reverse" architecture attempts to match the inherent driving current asymmetry of the access transistor to the asymmetric switching currents of the MTJ [93]. It is widely assumed that the trade-off between these two architectures depends solely upon the ratio of the critical writing currents $I_C(P \rightarrow AP)$ and $I_C(AP \rightarrow P)$. If this ratio is greater than 1, a "reverse-connected" architecture should be used. Otherwise, a "conventional" architecture should be used. However, this is not exactly true as this assumption fails to take into account both the V_{GS} and V_{DS} operating points of the access transistors during the write operation. For state-of-the-art technology nodes with supply voltages below $1V$, the cutoff point between these architectures is a writing current ratio closer to 1.5.

8.2.1.2 Shared

Access device "sharing" is one potential technique for increasing cell density. As shown in Figure 8.3, one access transistor is connected to multiple MTJ devices, with additional bitlines to support independent access. This also allows the access transistor to be sized up to provide higher write current while maintaining the same overall memory density. However,

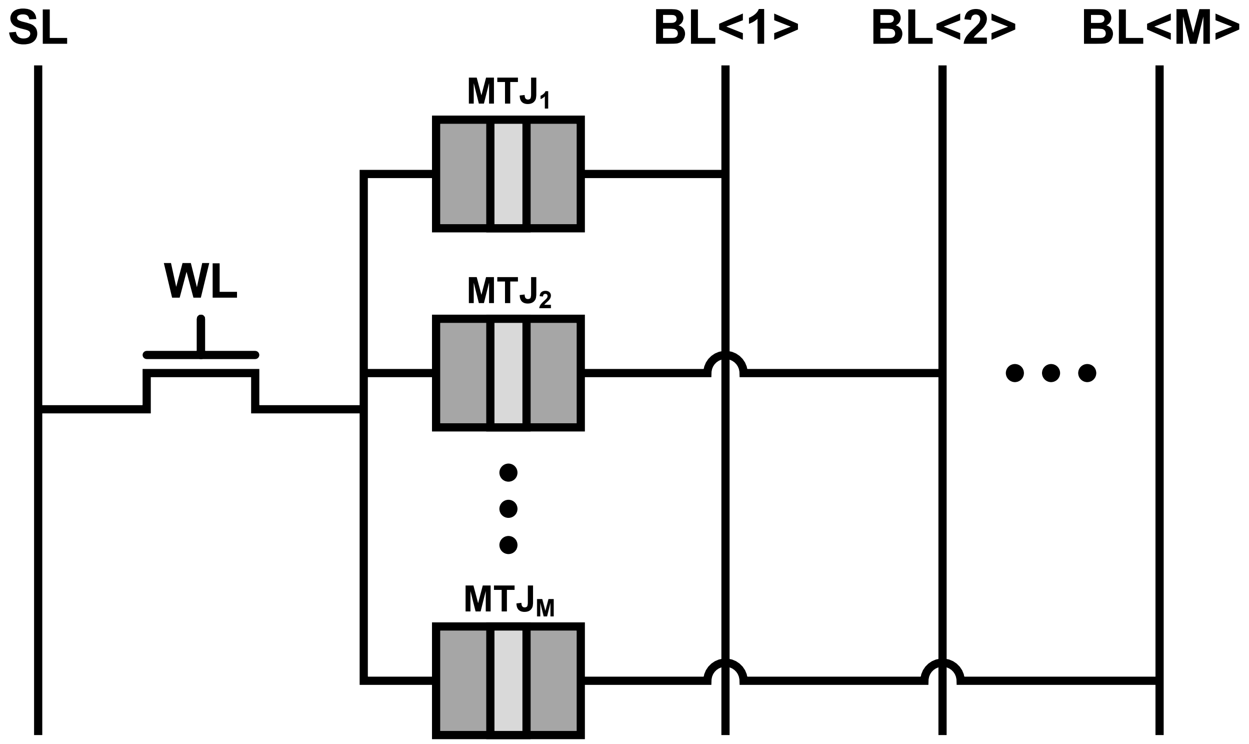


Figure 8.3: Shared memory cell architecture with M MTJs per transistor.

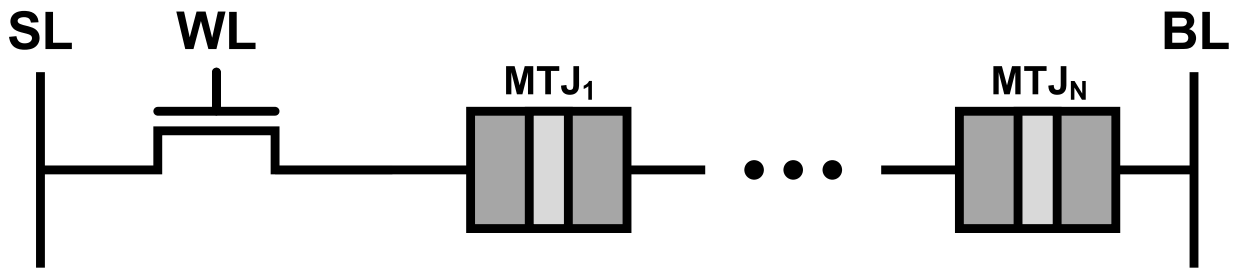


Figure 8.4: Stacked memory cell architecture with N MTJs per transistor.

there are several shortcomings associated with this technique which will be discussed in future sections.

8.2.1.3 Stacked

MTJ device "stacking" is another potential technique to increase cell density. "Stacking" works by connecting several different types of MTJ in series with one access transistor, shown in Figure 8.4, in a similar fashion to multi-bit Flash cells. To ensure functionality, the

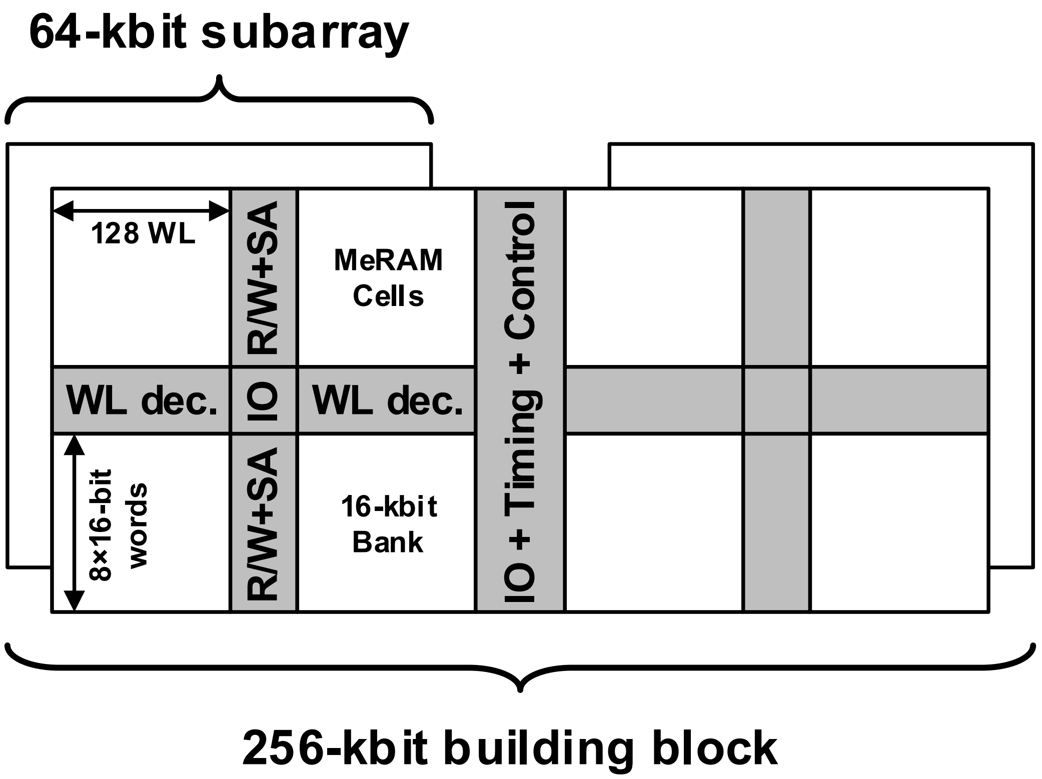


Figure 8.5: The 256-kbit building block with four 64-kbit subarrays. Each subarray is partitioned into four 16-kbit banks. Each bank has 128 word-lines, with eight 16-bit words per word-line.

resistance and critical writing current of each MTJ need to be sufficiently different. Reading and writing to a cell would require multiple cycles, one for each bit.

8.2.2 Subarraying

8.2.2.1 1T-1MTJ

As stated before, subarraying is necessary for larger memories. Single, large memory arrays are slow and require additional buffering to drive very long wires. Breaking it up into several smaller subarrays allows the memory to operate faster and share peripheral circuitry. For 1T-1MTJ cell architectures, the number of cells per bitline is limited by the capacitance of the access transistor and the MTJ itself. Generally, a single bitline can support no more than 256 cells.

8.2.2.2 Shared Architectures

As mentioned earlier, there are several shortcomings to a "shared" MTJ architecture. During the write operation, there are multiple parasitic current paths that siphon current from the device being written to, forcing the access device to be sized up [158]. These parasitic currents also have the potential to flip cells not being accessed. When reading, these parasitic paths lower the effective TMR that can be observed.

For M MTJs per access device and N wordlines per subarray, the effective TMR (TMR_{eff}) can be expressed as:

$$TMR_{eff} = \frac{2\,(N+M-1) - NM}{NM + (N-1)\,(M-1)\cdot TMR}\cdot TMR. \tag{8.1}$$

The worst case TMR degradation can be easily derived by finding the largest possible R_P and the smallest possible R_{AP}. If $R_{\|}$ is the parasitic parallel resistance shown in Figure 8.6, then $R_{P,MAX}$ can be calculated as $R_P \parallel R_{\|}$, where:

$$R_{\|} = \frac{N+M-1}{(N-1)\,(M-1)}\cdot(1+TMR)\cdot R_P \tag{8.2}$$

for the case of all parasitic resistances in the antiparallel state. Similarly, $R_{AP,MIN}$ can be calculated as $R_P\cdot(1+TMR) \parallel R_{\|}$, where:

$$R_{\|} = \frac{N+M-1}{(N-1)\,(M-1)}\cdot R_P \tag{8.3}$$

for the case of all parasitic resistances in the parallel state. For $M > 2$, TMR_{eff} is negative, limiting sharing to only two MTJs per access device.

In order to quantify the impact of sharing on the writing operation, the maximum allowable disturbance current a parasitic device can handle before a significant probability of

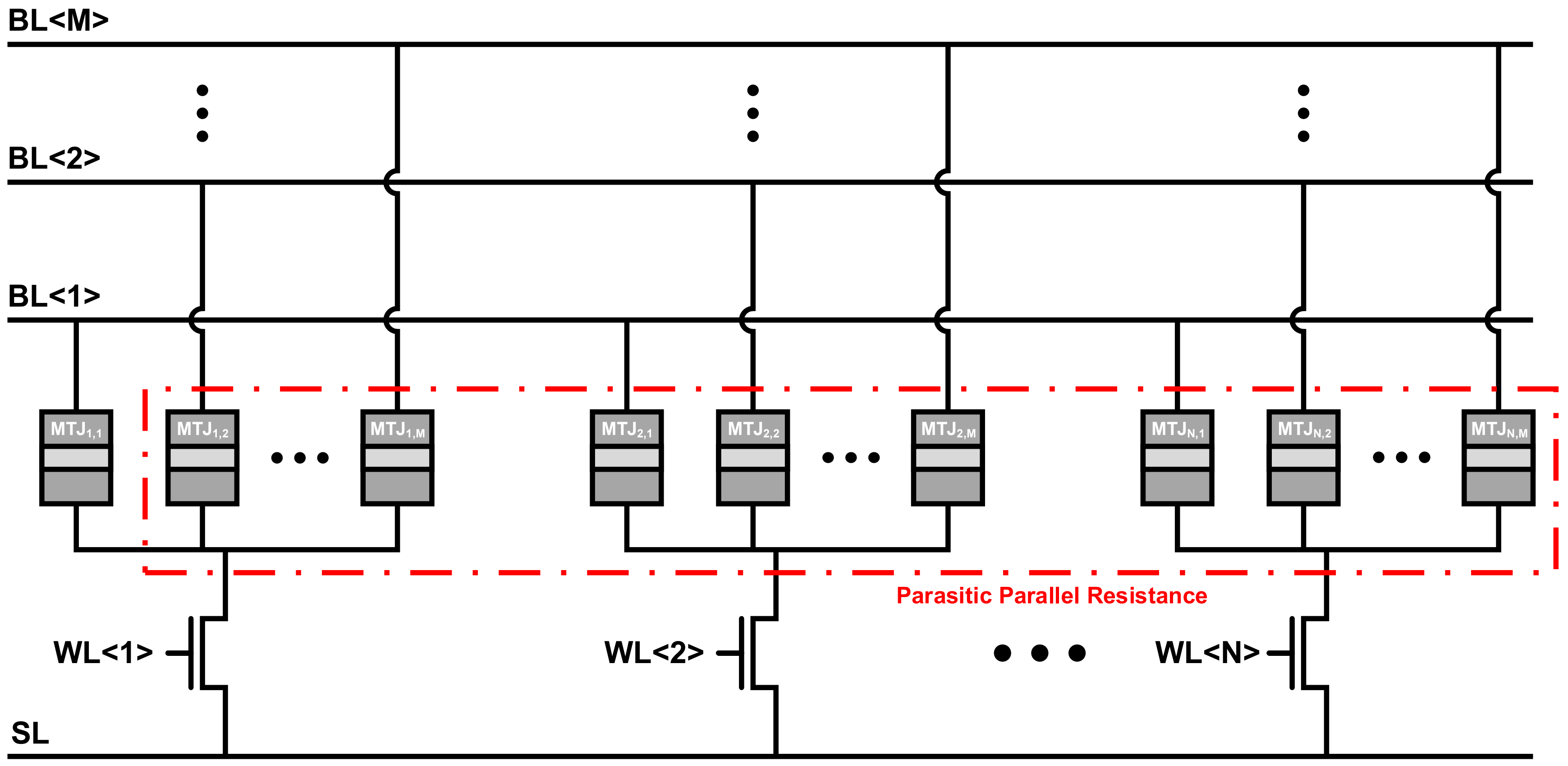
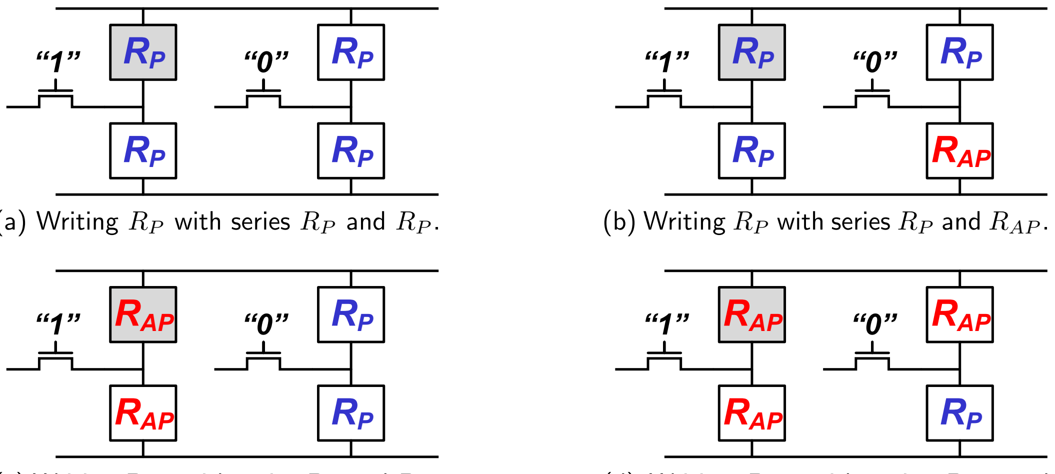


Figure 8.6: Shared architecture with M MTJs per transistor and N wordlines per subarray.



(a) Writing R_P with series R_P and R_P.

(b) Writing R_P with series R_P and R_{AP}.

(c) Writing R_{AP} with series R_P and R_P.

(d) Writing R_{AP} with series R_{AP} and R_P.

Figure 8.7: Worst-case writing configurations for sharing.

switching occurs must first be calculated. For $M = 2$, since the ability to read is already limited, the four worst corner cases are shown in Figure 8.7. For the corner case of Figure 8.7(a), we require:

$$R_P \cdot I_{WRITE}\left(P \to AP\right) - R_P \cdot I_{READ}\left(P \to AP\right) \leq 2R_P \cdot I_{READ}\left(P \to AP\right). \qquad (8.4)$$

Solving:

$$\frac{I_{READ}\left(P \to AP\right)}{I_{WRITE}\left(P \to AP\right)} \geq \frac{1}{3}. \qquad (8.5)$$

Similarly, for the other three cases, $I_{READ}/I_{WRITE} \geq 1/3$. This means that sharing can only be successfully implemented if the MTJ can tolerate a reading current greater than one third of the writing current without a significant probability of flipping.

8.3 STT-MRAM Memory Design

In this section, the design flow of three test chips implemented in 90nm, 65nm, and 45nm processes is described. Several architectures were selected for testing. Each design was subjected to the analysis outlined in [159], [160], and [161] in order to optimize read/write

Table 8.1: Time to read R_P (90nm)

R_P [Ω]	TMR [%]	Simulated [ns]	Measured [ns]	Cell Size[F^2]
500	100	3.77	7.20	30
500	100	3.17	5.20	55
670	50	7.30	8.50	30
670	50	5.18	10.2	55
670	200	3.04	4.20	30
670	200	2.69	4.80	55

performance, memory density, and energy considerations. Each chip was designed to operate with MTJs fabricated by UCLA's Western Institute of Nanoelectronics (WIN). MTJ device specifications are detailed in [162], [163] and [164].

8.3.1 90nm Bulk CMOS

Figure 8.8 shows a block diagram of the layout of our 90nm bulk CMOS test chip. The design work was performed for a conventional cell architecture and was intended to test the integration process flow. With a total of 6kbit, the memory had two 1kbit memory arrays, using RVT transistors with a cell size of $55F^2$, and two 2kbit memory arrays, using LVT transistors with a cell size of $30F^2$. For purposes of comparison, an SRAM cell in this technology is approximately $75F^2$. Included in the design are two resistor arrays with values ranging from a few hundred ohms to several kilohms (RSEL<16:0> and RSEL<33:17> in Figure 8.8), representing a range of TMR from 0% to 1000%. Simulated and measured results for the read performance of R_P (logical 0) are shown in Table 8.1. Similar results for the read performance of R_{AP} (logical 1) are shown in Table 8.2. The drive current of the $50F^2$ cell was also measured to be approximately $300\mu A$. Estimates show that this level of drive currentshould easily allow for thermally activated switching with write times on the order of 10 to 20 nanoseconds.

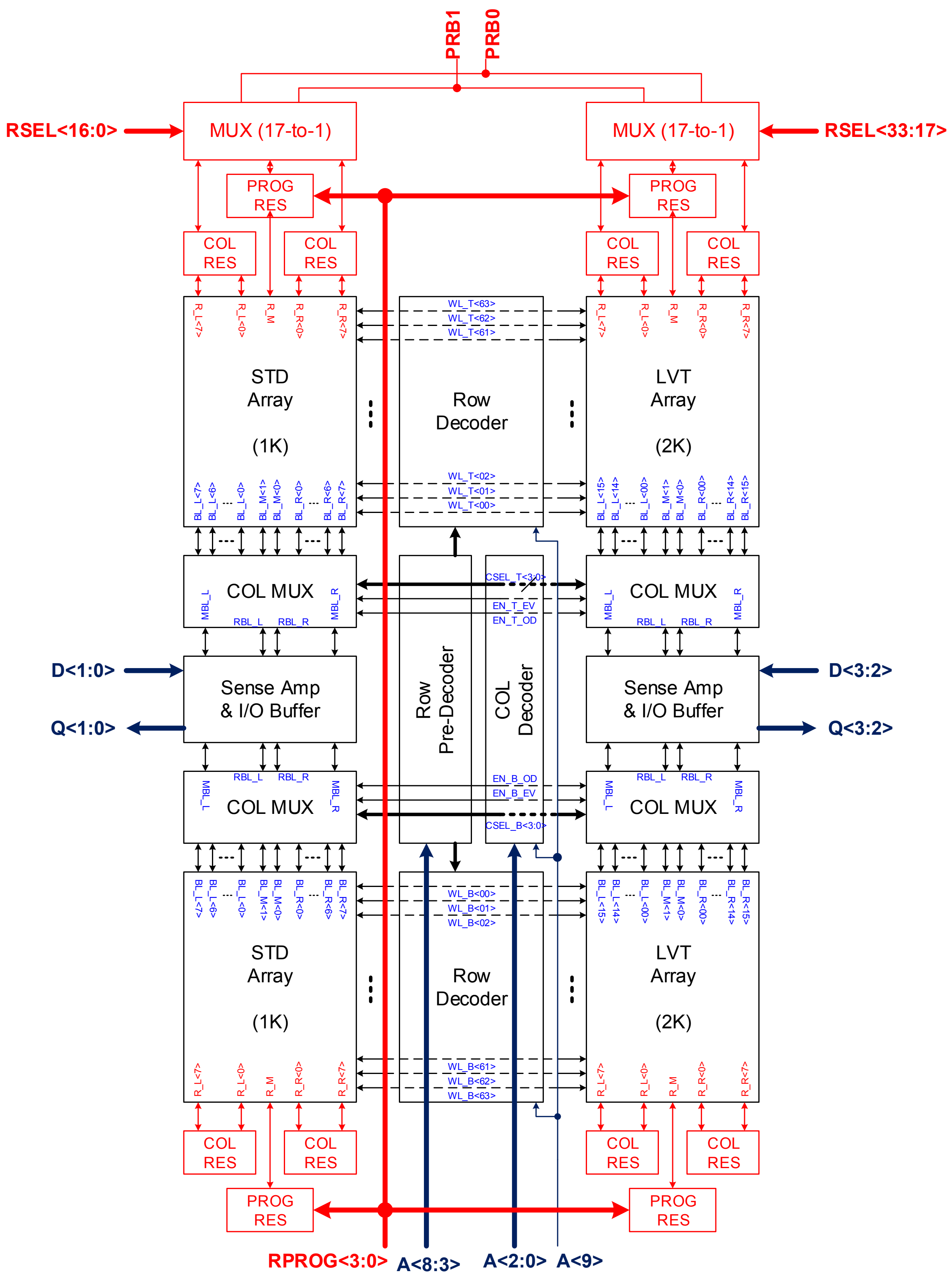


Figure 8.8: Block diagram of the 90nm STT-MRAM test chip.

Table 8.2: Time to read R_{AP} (90nm)

R_P [Ω]	TMR [%]	Simulated [ns]	Measured [ns]	Cell Size[F^2]
500	100	2.00	2.10	30 & 55
670	50	2.60	2.70	30 & 55
670	200	2.37	2.70	30 & 50

8.3.2 65nm Bulk CMOS

In the 65nm process, the memory array was increased to 16kbit and included three cell sizes: $28F^2$, $35F^2$, and $50F^2$. Again, the design work assumed a conventional cell architecture. A "short-pulse" reading scheme was also introduced with a bidirectional write driver (Figure 8.9) to improve read/write performance.

Short-pulse reading works by delivering a large, but very short, current pulse to the MTJ and latching in its value. A dual-wordline voltage boosting scheme (dual-boosting) was implemented that allowed the drive current to be increased by 70% in our smallest cell sizes and 130% in our largest cells. The resistor arrays from the 90nm design were kept for CMOS characterization. Figure 8.10 shows a layout of the 65nm design.

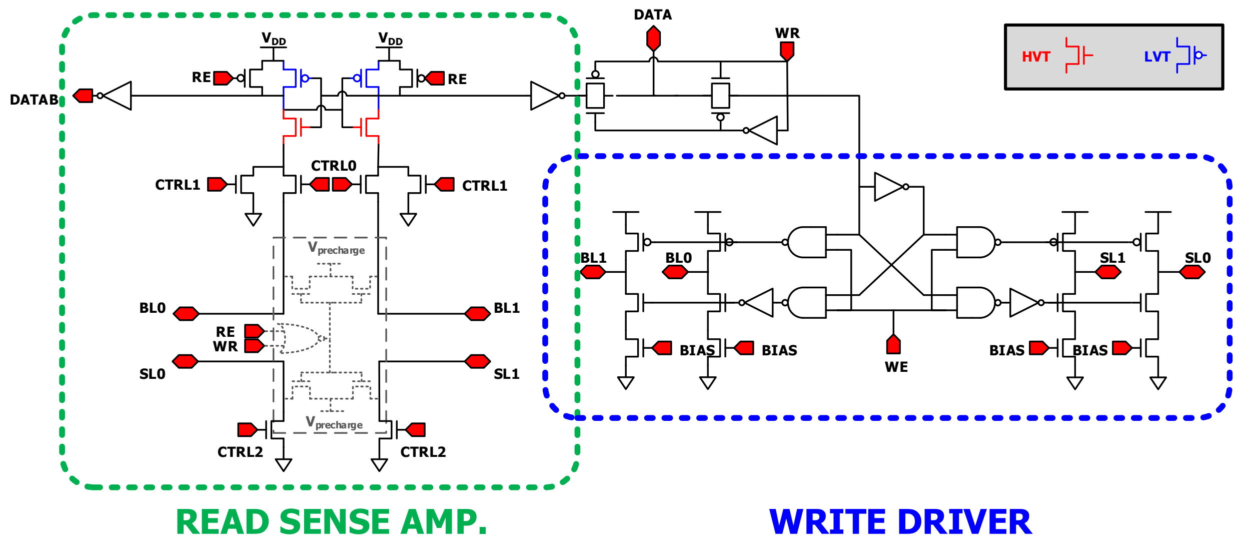


Figure 8.9: Read/Write driver for short-pulse reading.

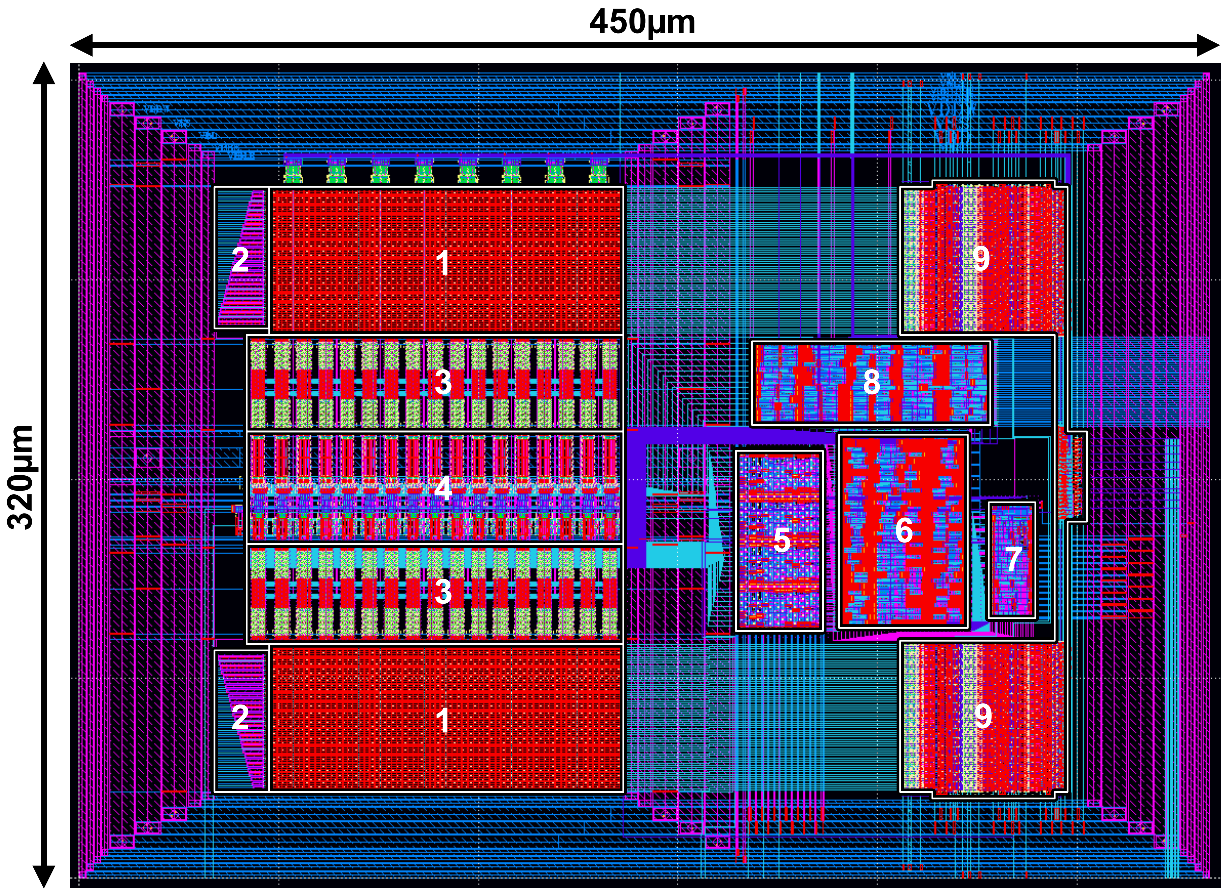


Figure 8.10: Cadence layout of the 65nm STT-MRAM test chip: (1) 8kbitmemory array, (2) resistor array, (3) muxes, (4) read/write circuitry, (5) read delay measurement circuitry, (6) configuration scan chain, (7) pulse generator, (8) data scan chain, and (9) decoder.

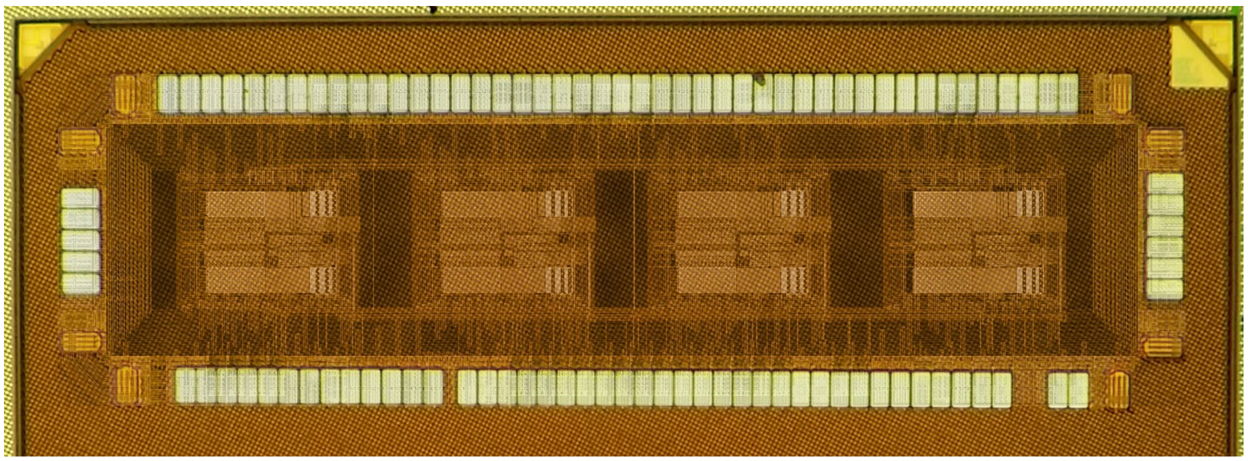


Figure 8.11: Chip micrograph of the 65nm STT-MRAM test chip.

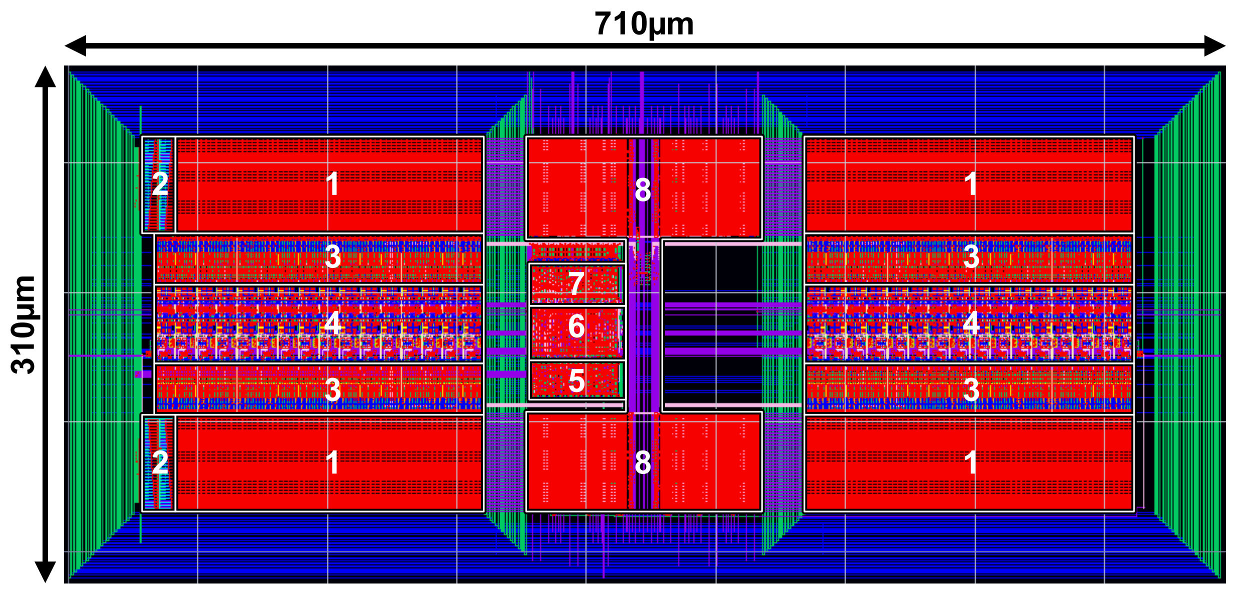


Figure 8.12: Cadence layout of the 45nm STT-MRAM test chip: (1) 8kbit memory array, (2) resistor array, (3) muxes, (4) read/write circuitry, (5) configuration scan chain, (6) pulse generator and read delay measurement circuitry, (7) data scan chain, and (8) decoder.

8.3.3 45nm SOI CMOS

The 45nm design moved the bulk CMOS to an SOI process. The memory size was increased to 32kbit and the same memory architecture was kept from our 65nm design (dual-boosting with short-pulse reading). It was possible to decrease the cell sizes to $17F^2$, $25F^2$, and $40F^2$, while still maintaining the same drive current as the 65nm design. Overall, large improvements to the memory layout and organization were made. Figure 8.12 shows a layout of the 45nm design.

8.3.4 Design Comparison

A comparison of the 90nm, 65nm and 45nm chips presented in this work is done with state-of-the-art STT-MRAMS from [156], [165], [166], and [167].

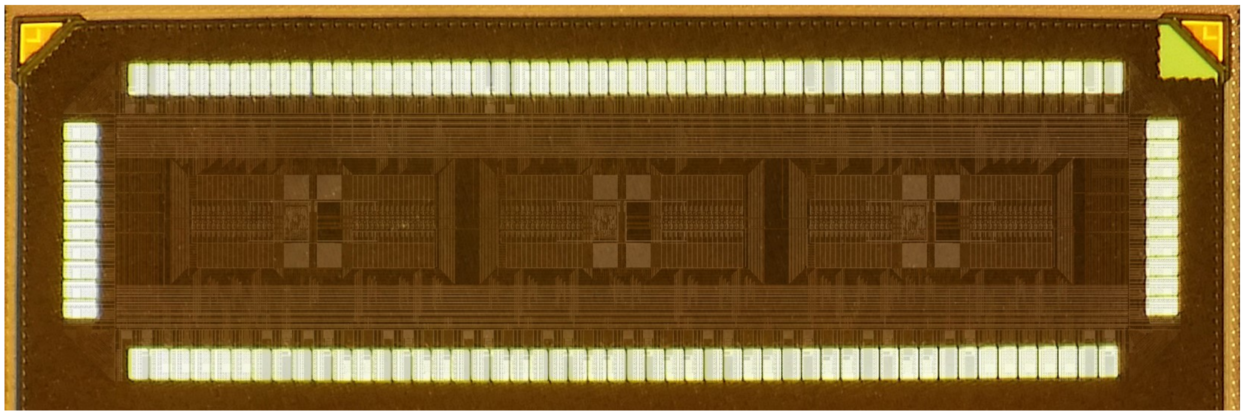


Figure 8.13: Chip micrograph of the 45nm STT-MRAM test chip.

8.4 MeRAM Memory Architectures

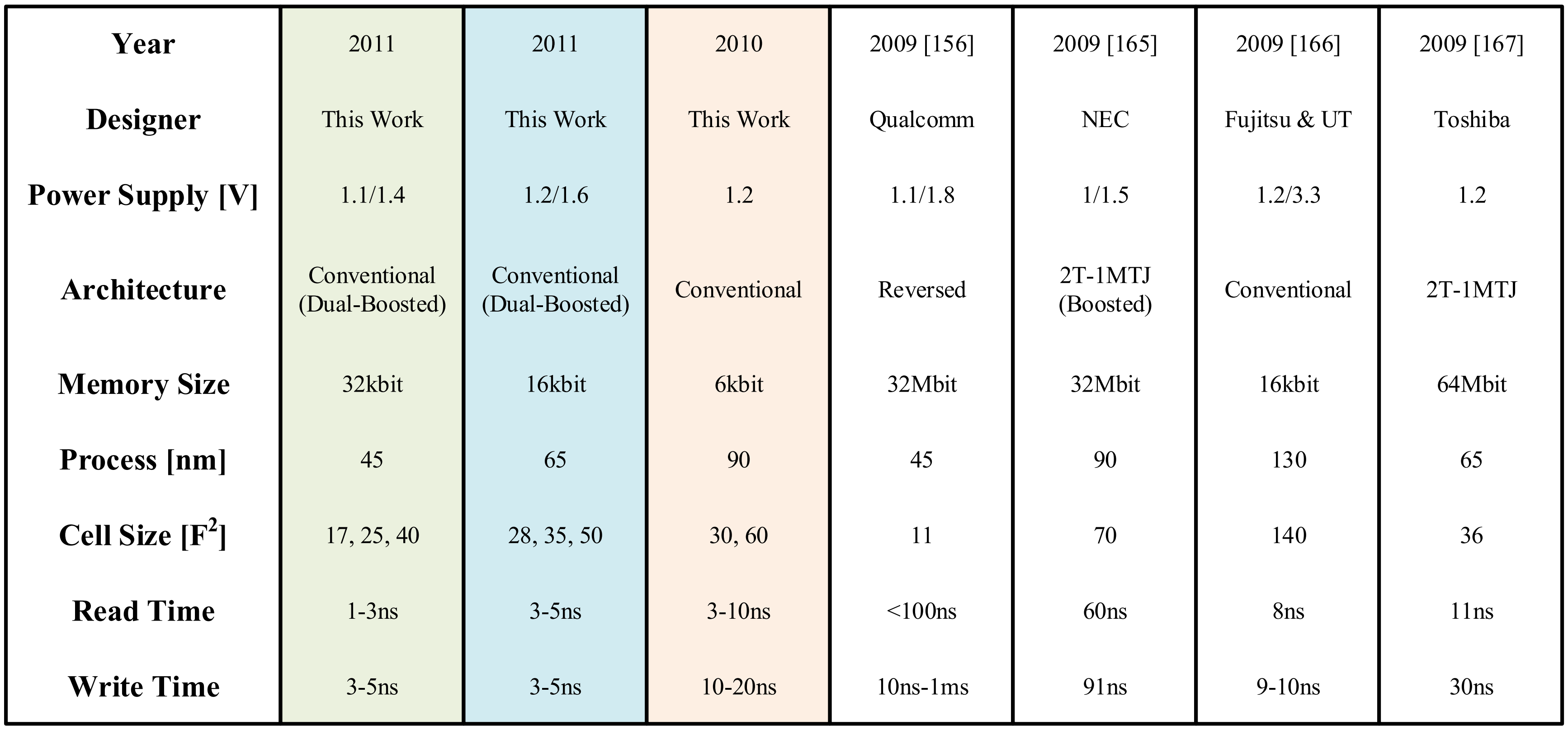
Two main types of memory architectures exist for MeRAMs. The first is similar to a 1T-1MTJ STT-MRAM architecture, but allows for a high speed read-disturbance-free operation. The second is 1D-MTJ Crossbar Array that allows for a sub-4F^2 cell size.

8.4.1 1T-1MTJ Array

The schematic of an array of MeRAM cells is shown in Figure 8.15, with multiple word-lines (WL_0, WL_1, WL_2, and WL_3) bit-lines (BL_A and BL_B), and a shared source-line (SL). Data is stored in 1T-1MTJ fashion in each cell. By sharing source-lines between adjacent MeRAM cells, a smaller cell size per bit can be achieved in a CMOS technology.

Due to the shared source-lines, multiple MeRAM cells can be written simultaneously during the write operation. To do this, SL is pulled low and the WL corresponding to the desired row of MeRAM cells is pulled high. MTJ_1 and MTJ_2 are then written by applying the appropriate switching voltages to BLA and BLB, respectively (see Figure 8.15).

Unfortunately, multiple MeRAM bits cannot also be read simultaneously during the read operation. To read MTJ_{0A}, WL_0 is pulled high, BL_A is grounded, and BL_B is left floating. A sense-amp is then used to measure the resistance of the SL. Similarly, to read MTJ_{0B}, WL_0 is pulled high, BL_B is grounded, and BL_A is left floating. Since the voltage applied between



Year	2011	2011	2010	2009 [156]	2009 [165]	2009 [166]	2009 [167]
Designer	This Work	This Work	This Work	Qualcomm	NEC	Fujitsu & UT	Toshiba
Power Supply [V]	1.1/1.4	1.2/1.6	1.2	1.1/1.8	1/1.5	1.2/3.3	1.2
Architecture	Conventional (Dual-Boosted)	Conventional (Dual-Boosted)	Conventional	Reversed	2T-1MTJ (Boosted)	Conventional	2T-1MTJ
Memory Size	32kbit	16kbit	6kbit	32Mbit	32Mbit	16kbit	64Mbit
Process [nm]	45	65	90	45	90	130	65
Cell Size [F^2]	17, 25, 40	28, 35, 50	30, 60	11	70	140	36
Read Time	1-3ns	3-5ns	3-10ns	<100ns	60ns	8ns	11ns
Write Time	3-5ns	3-5ns	10-20ns	10ns-1ms	91ns	9-10ns	30ns

Figure 8.14: Design comparison of STT-MRAMs.

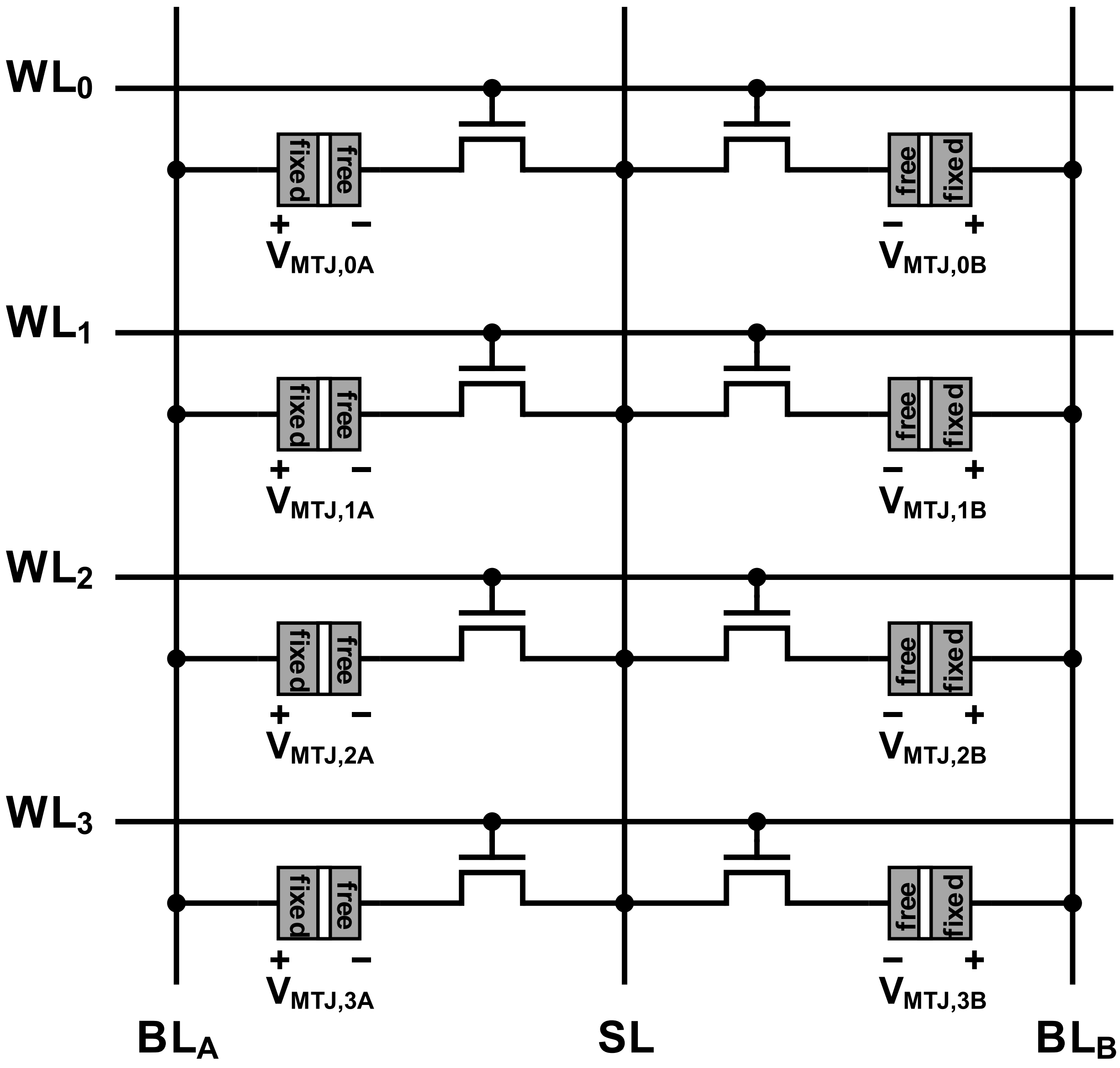


Figure 8.15: Schematic of an array of MeRAM cells using voltage-controlled MTJs. During a read, only negative voltages are applied between the fixed and free layers of the MTJs, creating read-disturbance-free operation. Positive voltages are applied during the write.

the fixed and free layers of both MTJs is negative during the entire read operation for both cases, the probability of accidental writing is 0.

In simulation, the MeRAM building block can run up to 1.2 GHz with a 1-cycle latency random access read and a 6-cycle random access write. A short-pulse-reading (SPR) sense-amplifier with body-voltage-based sensing is used to achieve sub-1ns reading performance [168] (see Figure 8.16). Burst writing can significantly reduce write overhead by exploiting

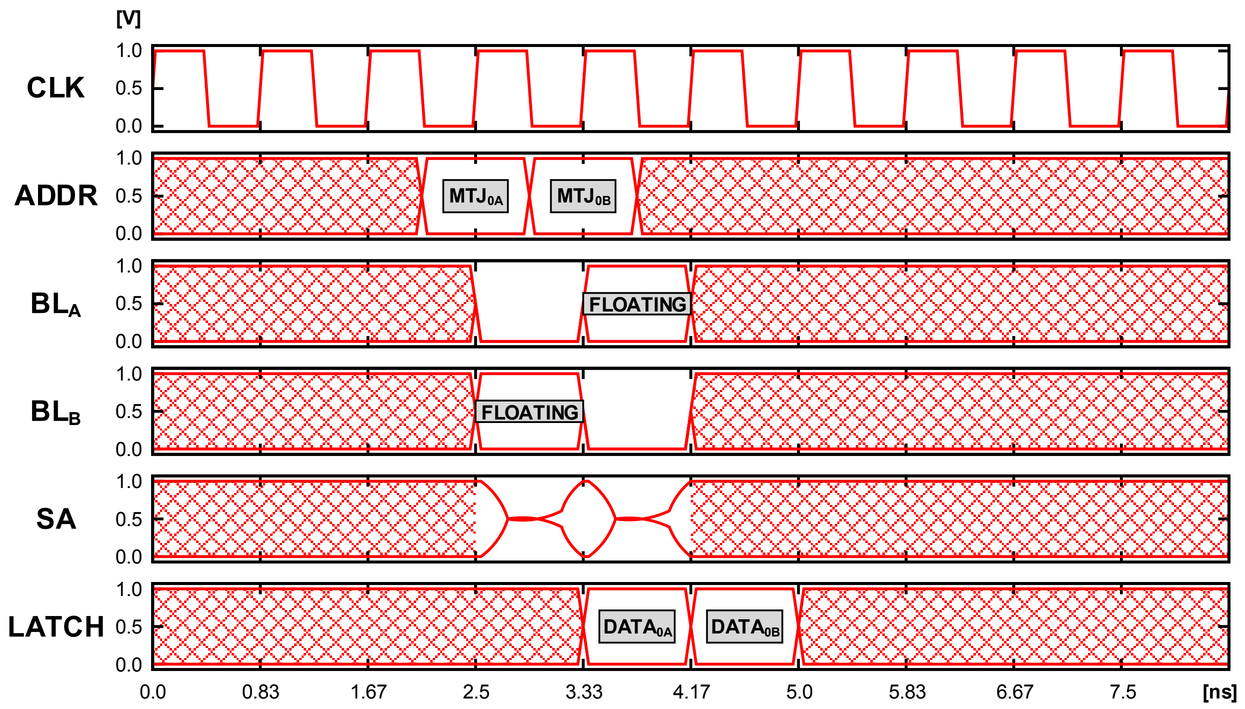


Figure 8.16: Timing diagram for a back-to-back, single-cycle latency read for bits $\mathrm{MTJ_{0A}}$ and $\mathrm{MTJ_{0B}}$ from Figure 8.15.

the ability to simultaneously write several MeRAM cells through the use of the shared source-lines. Writing in 2-, 4-, and 8-word bursts takes only 7, 9, and 13 cycles, respectively, and increases throughput between 70-370% (see Figure 8.17).

8.4.2 1D-MTJ Crossbar Array

Figures 8.18(a)-(b) show the schematic and layout view for one vertical slice of a high-density crossbar memory array using voltage-controlled MTJs. The unipolar set/reset write scheme of the devices allows for a diode to be integrated in series without any loss in functionality. The series diode also has the added benefit of eliminating the sneak currents present in traditional crossbar arrays [169]. Also, by eliminating the access transistor present in previous 1T-1MTJ designs, MeRAM can improve memory density with the 3D stacking of memory layers. A series stacked diode, fabricated on top of the MTJ, means that a $4F^2$ cell can be

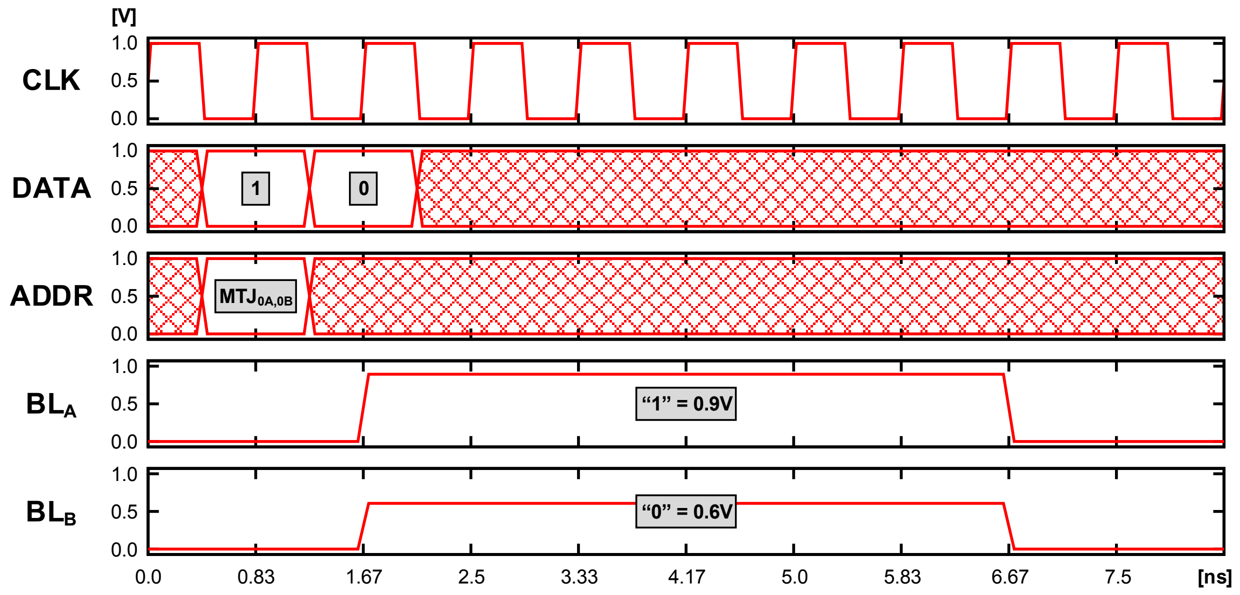


Figure 8.17: Timing diagram for a 2-word burst writing scheme. Writing takes 7 cycles to complete and increases throughput by 70%.

realized per MeRAM layer.

8.5 MeRAM Memory Design

In this section, the design flow of two MeRAM test chips are described. MTJ device specifications are detailed in [133] and [170].

8.5.1 1T-1MTJ Array

A 4kbit test chip for the 1T-1MTJ architecture outlined in [141] was fabricated in a 130nm CMOS technology. Figure 8.19 shows a layout view of a 4kbit 1T-1MTJ MeRAM array. The array measures 916.89μm by 1276.29μm and tests MTJ with a diameter of 100nm, 400nm, 1,000nm, and 2,000nm.

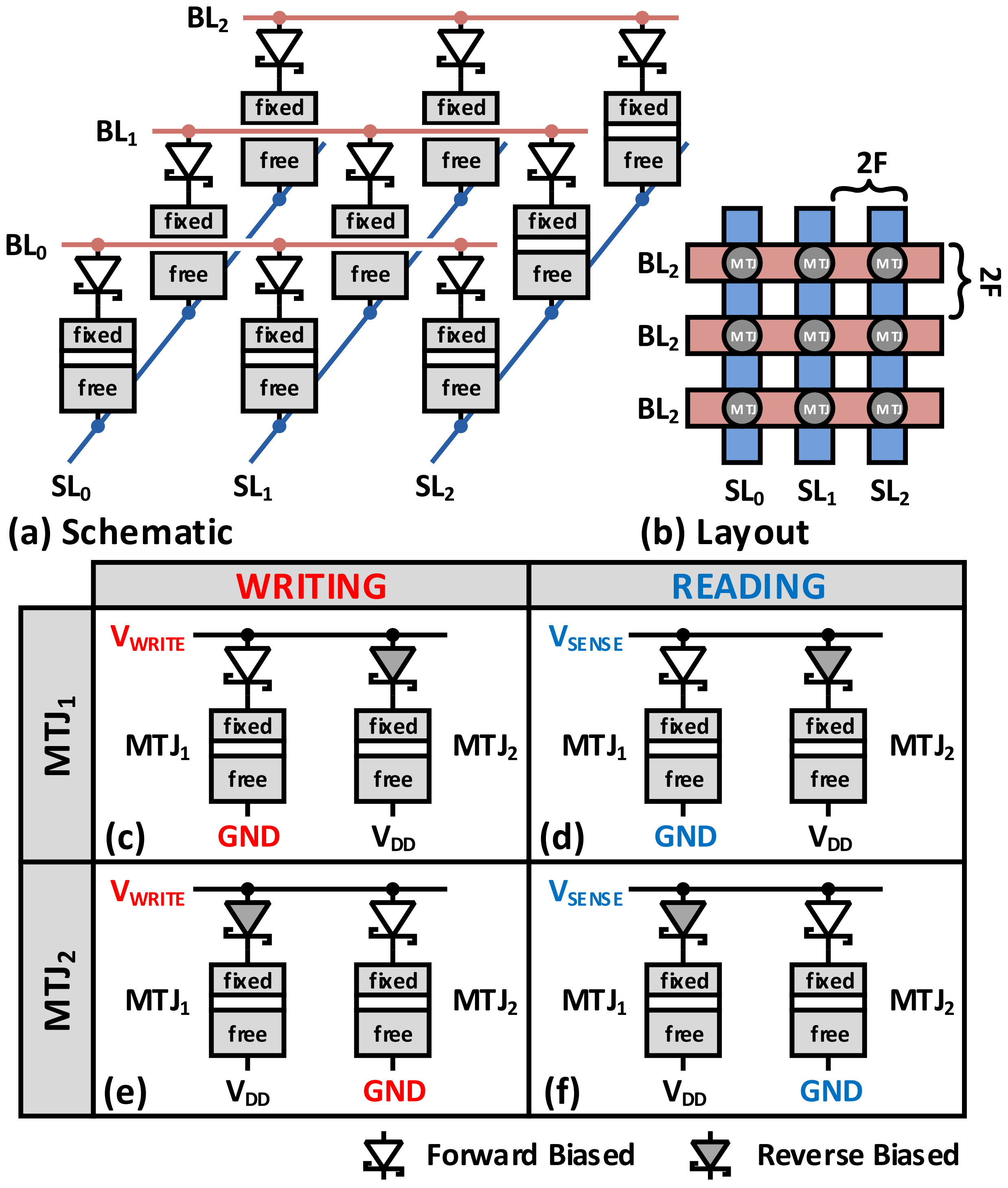


Figure 8.18: The (a) schematic and (b) layout views of the crossbar array structure, with integrated diodes, for 3D MeRAM. The array configuration for reading (d, f) and writing (c, e) data out of the testing setup.

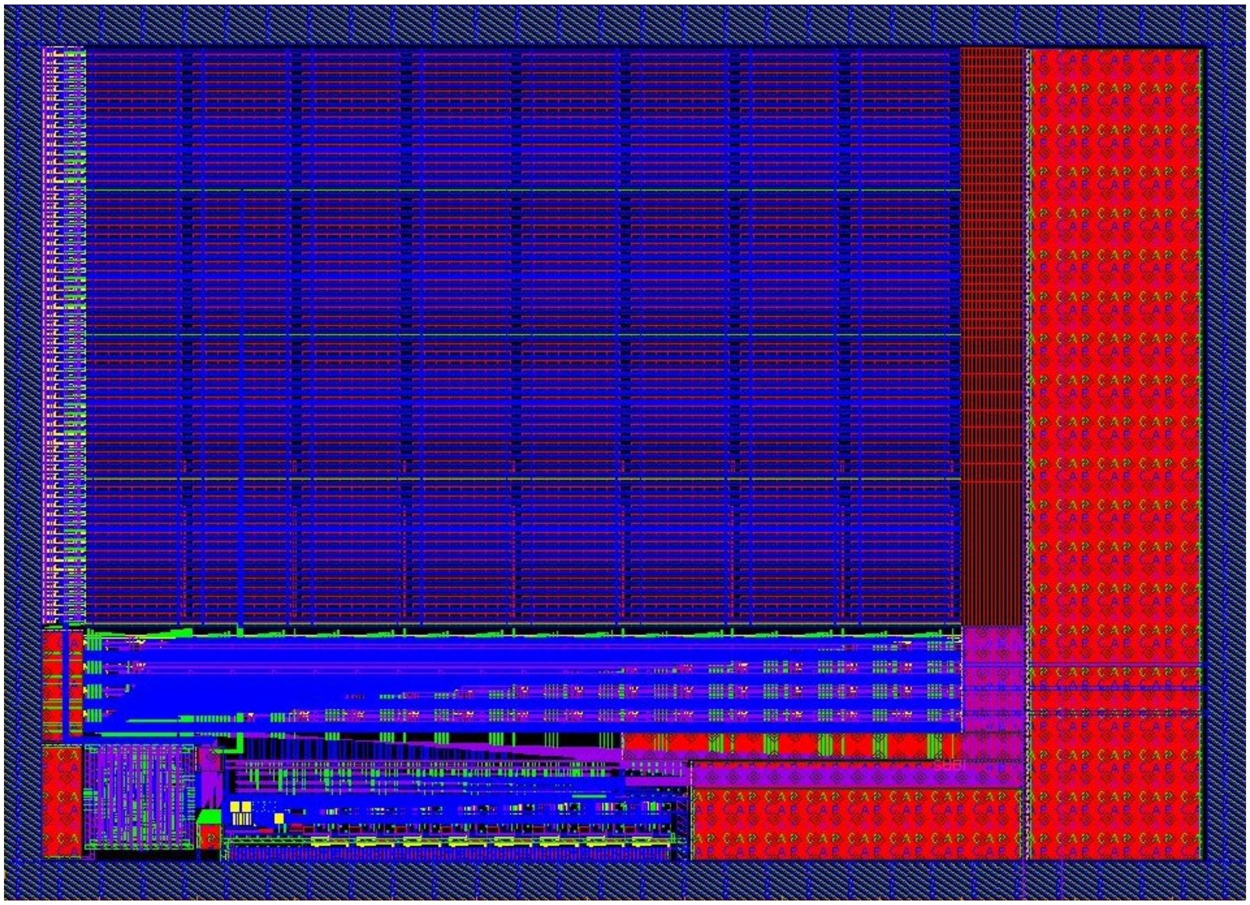
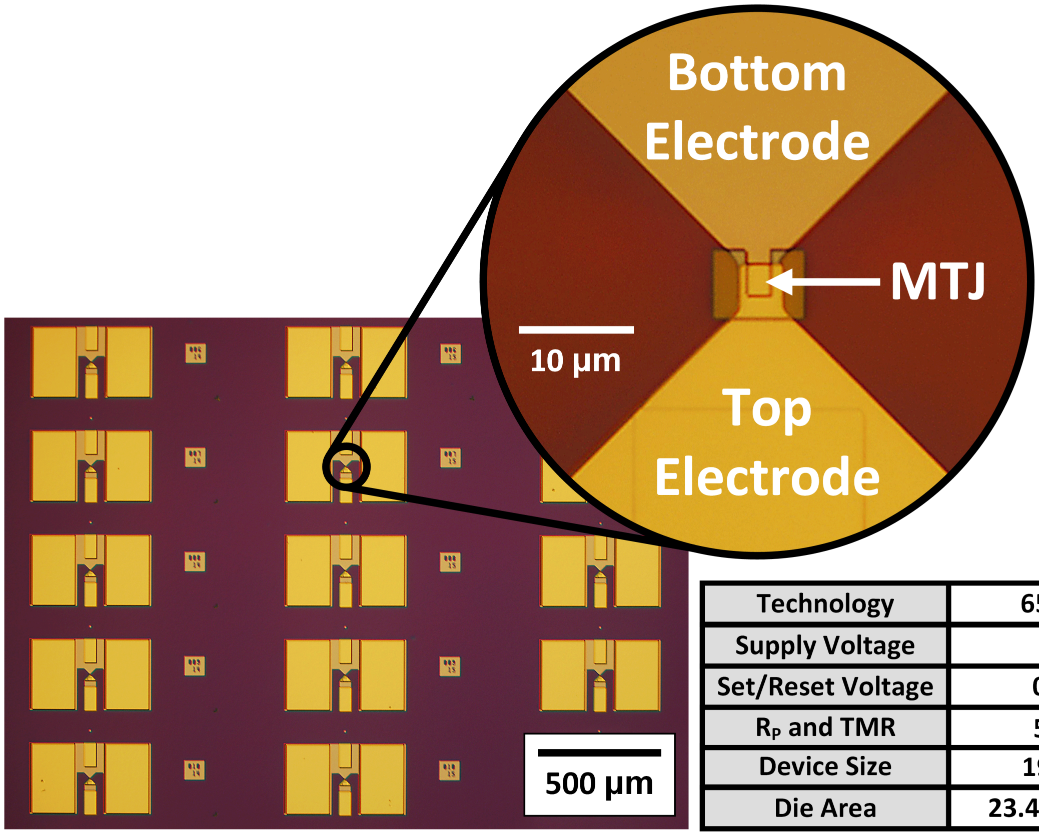


Figure 8.19: Layout view of a 4kbit 1T-1MTJ MeRAM array.

8.5.2 1D-MTJ Crossbar Array

A small crossbar memory array was constructed from 190 nm by 60 nm MTJ nanopillars (corresponding to a 65nm CMOS technology), with an R_P of 5.8kΩ, $H_{\text{Bias}} \approx +15$ Oe and thermal stability $\Delta = E/k_BT > 40$ (where E is the energy barrier between the two MTJ free layer states, k_B is the Boltzmann constant, and T is the operating temperature) for both P and AP states (projected to >10 years using magnetic-field-dependent switching measurements) at room temperature. The MTJs were connected to discrete germanium diodes (V_{TH} = 0.2V) used as access devices. The MTJ material stack was sputter deposited using a Singulus TIMARIS PVD system. Devices were fabricated using electron-beam lithography and ion milling techniques from the following stack: Ta (5) / $Co_{20}Fe_{60}B_{20}$ (1.1 – Free Layer) / MgO (1.2) / $Co_{60}Fe_{20}B_{20}$ (2.7) / Ru (0.85) / $Co_{70}Fe_{30}$ (2.3) / PtMn (20) (thickness in nm).



Technology	65-nm CMOS
Supply Voltage	1.5V
Set/Reset Voltage	0.5V / 1.1V
R_P and TMR	5.8kΩ / 5%
Device Size	190 x 60 nm^2
Die Area	23.43 x 13.05 mm^2

Figure 8.20: Chip micrograph of the fabricated MeRAM cells.

The thickness of the free layer was designed to be near the compensation between in-plane shape anisotropy and out-of-plane interface anisotropy in order to maximize the VCMA-induced manipulation over the state of the free layer. However, this compensation also leads to canting of the free layer due to the influence of higher-order anisotropy terms [130, 133], reducing the effective TMR between the equilibrium states of our MTJs to 5%. It is expected that further enhancement of the VCMA effect via materials optimization will mitigate the presence of canted states for our unipolar switching scheme and allow for fully in-plane or perpendicular magnetization with TMR >100%.

The probability of writing as a function of applied voltage for the MTJs used in the crossbar is shown in Fig. 7.8, demonstrating their unipolar set/reset characteristics. Data was obtained using 100 repetitions for each voltage with 100ms pulse widths. An overlap between the switching peaks is observed, which is expected to be reduced for larger VCMA effect values and would result in improved write noise-margin for the bit cell. Similarly,

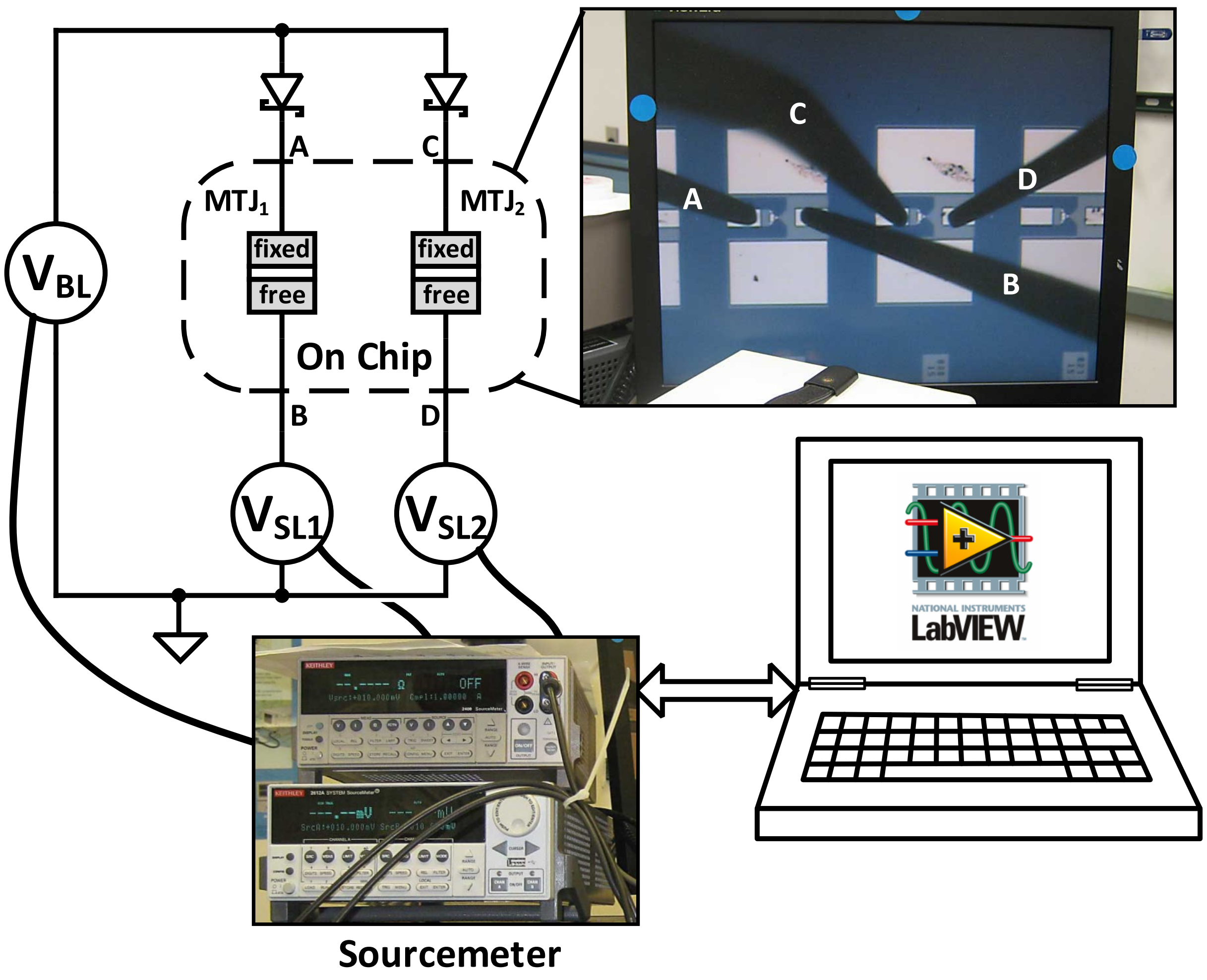


Figure 8.21: Testing setup for the MeRAM crossbar array.

reducing the reverse-bias diode leakage current improves the read noise-margin. The configuration of the testing setup for reading and writing the devices in the test array is shown in Figures 8.18(c)-(f). A schematic of the testing setup in shown in Figure 8.21. During both reading and writing, unaccessed bit-lines (BLs) are grounded while unaccessed source-lines (SLs) are pulled to V_{DD} (1.5V), reverse biasing the series diode for unaccessed bits. During the write operation, the target SL is pulled to ground, while the target BL is pulsed with the appropriate set/reset voltage, 0.5V and 1.1V respectively. During the read operation, the target SL is pulled to ground and the target BL is connected to a sense amplifier. To prevent disturbing the state of the desired bit cell, a sensing voltage of 0.2V is used. In order to maximize the read margin of the accessed memory cell, minimizing the forward drop across

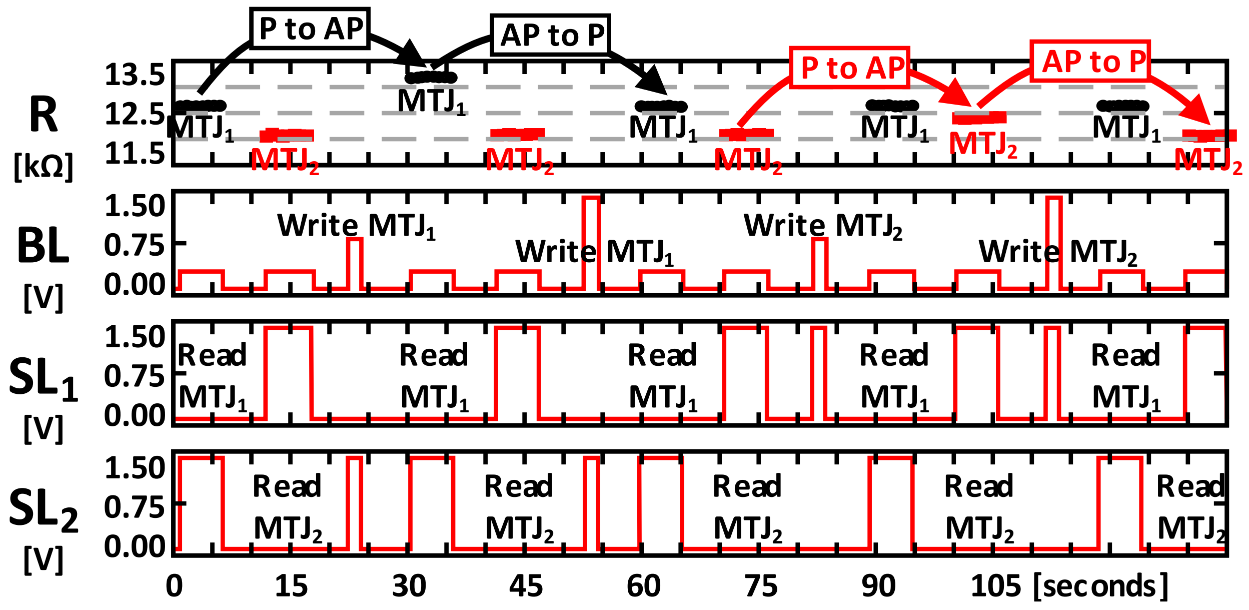


Figure 8.22: Measured transient waveforms for reading and writing the crossbar array, demonstrating that MTJ_1 can be written with unipolar set/reset voltages (0.5V and 1.1V respectively) and read without disturbing MTJ_2 and vice versa. A sensing voltage of 0.2V in the source-line is used for the read process, while unaccessed source-lines (SLs) are pulled to V_{DD} (1.5V).

the access diode is crucial. This is accomplished through the use of low-threshold Schottky (germanium) diodes. The ultimate size of the crossbar array is determined by the reverse bias leakage of the diodes and the resistance of the MTJs. Larger arrays can be built if the resistance of the MTJs is increased or if the leakage current of the diodes is reduced.

Figure 8.22 shows experimental transient waveforms demonstrating the functionality of the crossbar memory array. MTJ_1 and MTJ_2 are first initialized into the P state using an external magnetic field. Then, MTJ_1 is switched from P to AP, then back to P, without disturbing the value of MTJ_2. Similarly, MTJ_2 is also switched from P to AP, then back to P, without disturbing the value of MTJ_1. Switching is preformed using voltage pulses of 0.5V and 1.1V for a period of 1 second. After writing, both MTJ_1 and MTJ_2 are read 20 times using 0.2V without disturbing the state of the MTJs in the array.

CHAPTER 9

Conclusion

9.1 Research Contributions

Specific accomplishments of this research are:

- Analyzed the algorithm complexity and execution of level 1, 2, and 3 sparse-BLAS routines to theoretically derive the computational efficiency limits of CPU and GPU architectures. The resulting computational efficiency of CPUs and GPUs for sparse algorithms sits around 3% and 0.3%, and was experimentally confirmed.

- Developed an alternative method for performing sparse-BLAS that reduces the required memory bandwidth by 50%.

- Designed a scalable VLSI architecture that efficiently performs sparse-BLAS using the CSC data format, resulting in a 100% of utilization of the computing resources.

- Proposed a data stream reordering system to eliminate data hazards with minimal overhead. The resulting data "Shuffler" eliminated over 99% of data hazards in 14 test matrices for an average boost of 20% in computational efficiency.

- Evaluated the scalability and performance of sparse-BLAS kernel on a Virtex-5 SX95T FPGA. Compared to prior CPU, GPU, and FPGA implementations our design can boost the computational efficiency of sparse linear algebra by 54x, 322x, and 7x respectively. This results in an average improvement in energy efficiency of 96.1x, 48.9x, and 15.6x compared to traditional CPU, GPU, and FPGA architectures.

- Demonstrated a real-time throughput of up to 19.2 GFLOP/s for SpMxV on a Virtex-5 SX95T FPGA with a measured power of 5.1W. The resulting energy efficiency is more than a 50x and 38x improvement in energy efficiency over the CPU and GPU implementations, respectively.

- Prototyped a flexible sparse-BLAS kernel in a 40nm 1P10M CMOS process that can operate up to 515MHz with a core voltage between 0.55V and 1V. Running at 515MHz, the four PEs can achieve a maximum performance of 4.12 GFLOP/s.

- The minimum energy point, for a core voltage of 0.6V, running at 160MHz, and a measure power of 6.73mW, resulted in an energy efficiency of 190.31 GFLOP/s/W. This represents more than a 3,073x, 2,262x, and 66.6x improvement in energy efficiency over current state-of-the-art CPU, GPU, and FPGA implementations, respectively.
- The development of a physics-based MTJ macro-model capable of accurately modeling and predicting device behavior across temperature for both STT-MRAM and MeRAM. The model was implemented in Verilog-A and available in Appendix A.
- The design of three STT-MRAM test chips:
  1. 90nm bulk CMOS: 6kbit memory array, 2.5x density improvement over SRAM, 2-10ns read, $200\mu A$ write current.
  2. 65nm bulk CMOS: 16kbit memory array, 4.3x density improvement over SRAM, new high-speed read/write architecture, $400\mu A$ write current.
  3. 45nm SOI CMOS: 32kbit memory array, 7.1x density improvement over SRAM, average 3.8x density improvement over state-of-the-art STT-MRAMs, $500\mu A$ write current.
- The design of two MeRAM test chips:
  1. 1T-1MTJ Array: 130nm bulk CMOS 4kbit memory array, US patent US8988923 B2 [141].
  2. 1D-1MTJ Crossbar: discrete demonstration, potential for sub-1F^2 cell size.

9.2 Future Work

Areas of future work include:

- Further optimization and refinement of the FPGA-based sparse-BLAS kernel, including, but not limited to:

    (1) Integration with the Xilinx MicroBlaze soft-core processor. Including low level drivers in a Xilinx subset of C code to support sparse linear algebra subroutines. This enables backwards compatibility to existing code archives and abstracts away the programing interface to the processor as standard C code.
    (2) Implementation of the data stream reorder, a.k.a. the "Shuffler."
    (3) A refined and more expansive instruction set to better support sparse-BLAS routines.
    (4) Explicit support for sparse linear algebra solvers.

- Custom FPU design for the ASIC implementation to improve the throughput and every efficiency of the PE.
- Implementation of power gating in the PEs of the ASIC sparse-BLAS kernel for improved energy efficiency during idle periods.
- Refinement of the data stream reorder, a.k.a. the "Shuffler," to reduce the pipeline depth and overall overhead.
- Implementation of data compression and decompression on the CSC matrix data to further reduce the required memory bandwidth for sparse-BLAS.
- Further refinement of the MTJ model for improved accuracy and simulation times.
- Finalize the fabrication flow for MTJ/CMOS integration:
    (1) Performance and yield characterizations.
    (2) Integration of the Verilog-A model with a CAD process flow.
    (3) Development of a dual-ported STT-MRAM or MeRAM.

APPENDIX A

MTJ Verilog-A Model Code

```
//------------------------------------------------------------------//
// UCLA Electrical Engineering Department - MTJ Compact Model
//
// Author: Richard Dorrance
// Updated on July 1, 2015
//------------------------------------------------------------------//
`include "constants.vams"
`include "disciplines.vams"

module MTJ(p,n);

inout p,n;
electrical p,n;

//------------------------------------------------------------------//
// Input Parameters from the User
//------------------------------------------------------------------//
// MTJ length [m]
parameter   real        length          = 70e-9;
// MTJ width [m]
parameter   real        width           = 70e-9;
// MgO thickness [m]
parameter   real        dMgO            = 1.1e-9;
// Free layer thickness [m]
parameter   real        tfl             = 1.8e-9;
// Direct elastic tunnelling conductance [S/m^2]
parameter   real        Gt              = 8e+10;
// Conductance due to imperfections in Mgo [S/m^2/K^(4/3)]
parameter   real        Gsi             = 6e+6;
// Damping constant [unitless]
parameter   real        alpha           = 0.01;
// Spin-polarization at 0K [unitless]
```

```
parameter   real        P0              = 0.725;
// Spin-polarization temperature constant [K^(-3/2)]
parameter   real        a_sp            = 2e-5;
// Saturation magnetization [A/m]
parameter   real        Ms              = 1e+6;
// Temperature [K]
parameter   real        T               = 300;
// Time step for simulation [s]
parameter   real        t_step          = 1e-13;
// Initial state [0 = parallel, 1 = anti-parallel]
parameter   integer     initial_state   = 1 from [0:1];


//-------------------------------------------------------------------//
// Constants
//-------------------------------------------------------------------//
// Reduced Planck constant, [J*s]
`define hbar    (`P_H/`M_TWO_PI)
// Gyromagnetic ratio [m/(A x s)]
`define gamma   (221276/(1+pow(alpha,2)))
// External magnetic field x vector [A/m]
`define Ex      0
// External magnetic field y vector [A/m]
`define Ey      0
// External magnetic field z vector [A/m]
`define Ez      0
// Demagnetization factor in x direction
`define Nx      0.0045
// Demagnetization factor in y direction
`define Ny      0.0152
// Demagnetization factor in z direction
`define Nz      0.9803
// Anisotropy field constant [J/m^2]
`define Ki      0
// VCMA field constant [J/(V*m)]
`define Xi      0
// Field-like torque Beta1 parameter [unitless]
`define B1      0
// Field-like torque Beta2 parameter [1/A]
`define B2      0
// spin-polarization
`define P       (P0*(1-a_sp*sqrt(pow(T, 3))))
// MTJ area [m^2]
```

```
`define area    (`M_PI_4*length*width)
// MTJ volume [m^3]
`define Vol  (`area*tfl)
// Characteristic current density  [A/m^2]
`define Jc0     ((Ms*tfl*`P_Q*`P_U0)/`hbar)
// 4/3
`define fourthirds  1.33333333333333333333333333333


//------------------------------------------------------------------//
// Internal Variables
//------------------------------------------------------------------//
// MTJ resistance [Ohms]
 real   R;
 // Current density [A/m^2]
 real   J;
 // Intermediate normalized magnetic field components
 real   mx_int, my_int, mz_int;
 // Intermediate time derivative of magnetic field [1/s]
 real   dmx_int, dmy_int, dmz_int;
 // Intermediate denormalized intermediate magnetic field variables
 real   Mx_int, My_int, Mz_int;
 // Intermediate Heff components [A/m]
 real   Heffx_int, Heffy_int, Heffz_int;
 // Intermediate cross product components (m x p)
 real   mxpX_int, mxpY_int, mxpZ_int;
 // Intermediate cross product components (m x Heff)
 real   mxHeffx_int, mxHeffy_int, mxHeffz_int;
 // Normalized magnetic field components
 real   mx, my, mz;
 // Previous magnetic field components
 real   mx_old, my_old, mz_old;
 // Time derivative of magnetic field [1/s]
 real   dmx, dmy, dmz;
 // Previous time derivative of magnetic field [1/s]
 real   dmx_old, dmy_old, dmz_old;
 // Denormalized intermediate magnetic field variables
 real   Mx, My, Mz;
 // Heff components [A/m]
 real   Heffx, Heffy, Heffz;
 // Previous Heff components [A/m]
 real   Heffx_old, Heffy_old, Heffz_old;
 // Cross product components (m x p)
 real   mxpX, mxpY, mxpZ;
 // Cross product components (m x Heff)
 real   mxHeffx, mxHeffy, mxHeffz;
 // Gaussian random variables with mean = 0, stdev = 1
 real   randomX, randomY, randomZ;
 // Normalized thermal noise vector components
 real   sigmaX, sigmaY, sigmaZ;
 // Seed variables for RNG
 integer    seedX, seedY, seedZ;


//-----------------------------------------------------------------//
// Define resistance's relationship to the magnetic field
//-----------------------------------------------------------------//
analog function real getRes;
input mx;
real mx;
begin
    getRes = 1/(Gt*(1+mx*pow(`P, 2))+Gsi*pow(T, `fourthirds))/`area;
end
endfunction


//-----------------------------------------------------------------//
// Define the components of Heff
//-----------------------------------------------------------------//
analog function real getHeffx;
input mx, sigmaX, vmtj;
real mx, sigmaX, vmtj;
real Hext, Hdem, Han, Hvcma, Hth;
begin
    Hext = `Ex;
    Hdem = -Ms*`Nx*mx;
    Han = 0;
    Hvcma = 0;
    Hth = sigmaX*sqrt((2*`P_K*T*alpha)/(`P_U0*`gamma*Ms*`Vol*t_step));
    getHeffx = Hext + Hdem + Han + Hvcma + Hth;
 end
 endfunction

analog function real getHeffy;
input my, sigmaY, vmtj;
real my, sigmaY, vmtj;
real Hext, Hdem, Han, Hvcma, Hth;
begin
    Hext = `Ey;
    Hdem = -Ms*`Ny*my;
    Han = 0;
    Hvcma = 0;
    Hth = sigmaY*sqrt((2*`P_K*T*alpha)/(`P_U0*`gamma*Ms*`Vol*t_step));
    getHeffy = Hext + Hdem + Han + Hvcma + Hth;
 end
 endfunction

analog function real getHeffz;
input mz, sigmaZ, vmtj;
real mz, sigmaZ, vmtj;
real Hext, Hdem, Han, Hvcma, Hth;
begin
    Hext = `Ez;
    Hdem = -Ms*`Nz*mz;
    Han = (2*`Ki*mz)/(tfl*`P_U0*Ms);
    Hvcma = -(2*mz*`Xi*vmtj)/(`P_U0*Ms*dMgO*tfl);
    Hth = sigmaZ*sqrt((2*`P_K*T*alpha)/(`P_U0*`gamma*Ms*`Vol*t_step));
    getHeffz = Hext + Hdem + Han + Hvcma + Hth;
end
endfunction


//-------------------------------------------------------------------//
// Define cross(A, B)
//-------------------------------------------------------------------//
analog function real crossX;
input a1, a2, a3, b1, b2, b3;
real a1, a2, a3, b1, b2, b3;
begin
    crossX = a2*b3-a3*b2;
end
endfunction

analog function real crossY;
input a1, a2, a3, b1, b2, b3;
real a1, a2, a3, b1, b2, b3;
begin
    crossY = a3*b1-a1*b3;
end
```

```
endfunction

analog function real crossZ;
input a1, a2, a3, b1, b2, b3;
real a1, a2, a3, b1, b2, b3;
begin
    crossZ = a1*b2-a2*b1;
end
endfunction


//---------------------------------------------------------------------//
// Define mag(x, y, z)
//---------------------------------------------------------------------//
analog function real mag;
input x, y, z;
real x, y, z;
begin
    mag = sqrt(pow(x,2)+pow(y,2)+pow(z,2));
end
endfunction


//---------------------------------------------------------------------//
// LLG equations with STT
//---------------------------------------------------------------------//

 analog function real LLGx;
  //-----------------------------------------------------------------//
  // Inputs
  //-----------------------------------------------------------------//
  // Magnetic field
  input mx, my, mz;
  // Components of Heffective
  input Heffx, Heffy, Heffz;
  // Intermediate cross product components (m x Heff)
  input mxHeffx, mxHeffy, mxHeffz;
  // Intermediate cross product components (m x p)
  input mxpX, mxpY, mxpZ;
  // Current density
  input J;

  real mx, my, mz;
  real Heffx, Heffy, Heffz;
  real mxHeffx, mxHeffy, mxHeffz;
  real mxpX, mxpY, mxpZ;
  real J;

  //---------------------------------------------------------------//
  // Internal Variables
  //---------------------------------------------------------------//
  // Landau-Lifshitz term
  real Landau_Lifshitz;
  // Gilbert damping term
  real damping;
  // STT variables
  real STT;
  // Field-like torque term
  real FLT;

  begin

      //-------------------------------------------------------//
      // Calculate the damping component of the LLG equation.
      //-------------------------------------------------------//
      damping = alpha*crossX(mx, my, mz, mxHeffx, mxHeffy, mxHeffz);

      //-------------------------------------------------------//
      // Calculate the original Landau-Lifshitz component of
      // the LLG equation.
      //-------------------------------------------------------//
      Landau_Lifshitz = crossX(mx, my, mz, Heffx, Heffy, Heffz);

      //-------------------------------------------------------//
      // Calculate the STT component of the LLG equation.
      //-------------------------------------------------------//
      STT = (J/`Jc0)*`P*crossX(mx, my, mz, mxpX, mxpY, mxpZ);

      //-------------------------------------------------------//
      // Calculate the FLT component of the LLG equation.
      //-------------------------------------------------------//
      FLT = (J/`Jc0)*`P*(`B1+`B2*`area*J)*crossX(mx,my,mz,-1,0,0);

      //-------------------------------------------------------//
      // Calculate the X-component of the solution to the LLG
      // equation.
```

```
     //-----------------------------------------------------//
     LLGx = -`gamma*(Landau_Lifshitz + damping - STT - FLT);

 end
endfunction

analog function real LLGy;
 //-------------------------------------------------------------//
 // Inputs
 //-------------------------------------------------------------//
 // Magnetic field
 input mx, my, mz;
 // Components of Heffective
 input Heffx, Heffy, Heffz;
 // Intermediate cross product components (m x Heff)
 input mxHeffx, mxHeffy, mxHeffz;
 // Intermediate cross product components (m x p)
 input mxpX, mxpY, mxpZ;
 // Current density
 input J;

 real mx, my, mz;
 real Heffx, Heffy, Heffz;
 real mxHeffx, mxHeffy, mxHeffz;
 real mxpX, mxpY, mxpZ;
 real J;

 //-------------------------------------------------------------//
 // Internal Variables
 //-------------------------------------------------------------//
 // Landau-Lifshitz term
 real Landau_Lifshitz;
 // Gilbert damping term
 real damping;
 // STT variables
 real STT;
 // Field-like torque term
 real FLT;

 begin

     //-----------------------------------------------------//
     // Calculate the damping component of the LLG equation.
```

```
        //------------------------------------------------------//
        damping = alpha*crossY(mx, my, mz, mxHeffx, mxHeffy, mxHeffz);

        //------------------------------------------------------//
        // Calculate the original Landau-Lifshitz component of
        // the LLG equation.
        //------------------------------------------------------//
        Landau_Lifshitz = crossY(mx, my, mz, Heffx, Heffy, Heffz);

        //------------------------------------------------------//
        // Calculate the STT component of the LLG equation.
        //------------------------------------------------------//
        STT = (J/`Jc0)*`P*crossY(mx, my, mz, mxpX, mxpY, mxpZ);

        //------------------------------------------------------//
        // Calculate the FLT component of the LLG equation.
        //------------------------------------------------------//
        FLT = (J/`Jc0)*`P*(`B1+`B2*`area*J)*crossY(mx,my,mz,-1,0,0);

        //------------------------------------------------------//
        // Calculate the Y-component of the solution to the LLG
        // equation.
        //------------------------------------------------------//
        LLGy = -`gamma*(Landau_Lifshitz + damping - STT - FLT);

    end
endfunction

analog function real LLGz;
 //------------------------------------------------------------//
 // Inputs
 //------------------------------------------------------------//
 // Magnetic field
 input mx, my, mz;
 // Components of Heffective
 input Heffx, Heffy, Heffz;
 // Intermediate cross product components (m x Heff)
 input mxHeffx, mxHeffy, mxHeffz;
 // Intermediate cross product components (m x p)
 input mxpX, mxpY, mxpZ;
 // Current density
 input J;

```

```
real mx, my, mz;
real Heffx, Heffy, Heffz;
real mxHeffx, mxHeffy, mxHeffz;
real mxpX, mxpY, mxpZ;
real J;

//--------------------------------------------------------------//
// Internal Variables
//--------------------------------------------------------------//
// Landau-Lifshitz term
real Landau_Lifshitz;
// Gilbert damping term
real damping;
// STT variables
real STT;
// Field-like torque term
real FLT;

begin

    //------------------------------------------------------//
    // Calculate the damping component of the LLG equation.
    //------------------------------------------------------//
    damping = alpha*crossZ(mx, my, mz, mxHeffx, mxHeffy, mxHeffz);

    //------------------------------------------------------//
    // Calculate the original Landau-Lifshitz component of
    // the LLG equation.
    //------------------------------------------------------//
    Landau_Lifshitz = crossZ(mx, my, mz, Heffx, Heffy, Heffz);

    //------------------------------------------------------//
    // Calculate the STT component of the LLG equation.
    //------------------------------------------------------//
    STT = (J/`Jc0)*`P*crossZ(mx, my, mz, mxpX, mxpY, mxpZ);

    //------------------------------------------------------//
    // Calculate the FLT component of the LLG equation.
    //------------------------------------------------------//
    FLT = (J/`Jc0)*`P*(`B1+`B2*`area*J)*crossZ(mx,my,mz,-1,0,0);

    //------------------------------------------------------//
    // Calculate the Z-component of the solution to the LLG
```

```
      // equation.
      //------------------------------------------------------//
      LLGz = -`gamma*(Landau_Lifshitz + damping - STT - FLT);

  end
 endfunction


//--------------------------------------------------------------------//
// Analog Block: Defines the Behavior of the MTJ
//--------------------------------------------------------------------//
 analog begin

//----------------------------------------------------------------//
// Initialize the MTJ and Bound Transient Analysis Step Time
//----------------------------------------------------------------//
$bound_step(t_step);

@(initial_step) begin
    $strobe("-------------------------------");
    $strobe("MTJ and Simulation Specs");
    $strobe("-------------------------------");
    $strobe("TMR    = ", (getRes(-1)-getRes(1))/getRes(1));
    $strobe("Rp     = ", getRes(1));
    $strobe("Rap    = ", getRes(-1));
    $strobe("Jc0    = ", `Jc0);
    $strobe("-------------------------------");

    seedX   = $random;
    seedY   = $random;
    seedZ   = $random;

    if(initial_state) begin
        mx  = 1;
        my  = 0;
        mz  = 0;
    end else begin
        mx  = -1;
        my  = 0;
        mz  = 0;
    end

    R       = getRes(mx);
    Heffx   = 0;
    Heffy   = 0;
    Heffz   = 0;

end

//--------------------------------------------------------------//
// Define the MTJ to act like a resistor
//--------------------------------------------------------------//
V(p,n) <+ R*I(p,n);

//--------------------------------------------------------------//
// Backup/Define the necessary variables to preform a time step
//--------------------------------------------------------------//
 J           = I(p,n)/`area;

 mx_old      = mx;
 my_old      = my;
 mz_old      = mz;

 dmx_old     = dmx;
 dmy_old     = dmy;
 dmz_old     = dmz;

 Heffx_old   = Heffx;
 Heffy_old   = Heffy;
 Heffz_old   = Heffz;

//--------------------------------------------------------------//
// Use the LLG equation w/ Heun's Method to update the Magnetic Field
//--------------------------------------------------------------//
 mxpX_int        = crossX(mx_old, my_old, mz_old, -1, 0, 0);
 mxpY_int        = crossY(mx_old, my_old, mz_old, -1, 0, 0);
 mxpZ_int        = crossZ(mx_old, my_old, mz_old, -1, 0, 0);

 mxHeffx_int     = crossX(mx_old, my_old, mz_old, \
                           Heffx_old, Heffy_old, Heffz_old);
 mxHeffy_int     = crossY(mx_old, my_old, mz_old, \
                           Heffx_old, Heffy_old, Heffz_old);
 mxHeffz_int     = crossZ(mx_old, my_old, mz_old, \
                           Heffx_old, Heffy_old, Heffz_old);

 dmx_int     = LLGx(mx_old, my_old, mz_old, Heffx_old, Heffy_old, \
                     Heffz_old, mxHeffx_int, mxHeffy_int, \
             mxHeffz_int, mxpX_int, mxpY_int, mxpZ_int, J);
 dmy_int    = LLGy(mx_old, my_old, mz_old, Heffx_old, Heffy_old, \
                     Heffz_old, mxHeffx_int, mxHeffy_int, \
             mxHeffz_int, mxpX_int, mxpY_int, mxpZ_int, J);
 dmz_int    = LLGz(mx_old, my_old, mz_old, Heffx_old, Heffy_old, \
                     Heffz_old, mxHeffx_int, mxHeffy_int, \
             mxHeffz_int, mxpX_int, mxpY_int, mxpZ_int, J);

 Mx_int     = mx_old + (dmx_old*t_step);
 My_int     = my_old + (dmy_old*t_step);
 Mz_int     = mz_old + (dmz_old*t_step);

 mx_int     = Mx_int/mag(Mx_int, My_int, Mz_int);
 my_int     = My_int/mag(Mx_int, My_int, Mz_int);
 mz_int     = Mz_int/mag(Mx_int, My_int, Mz_int);

 randomX    = $rdist_normal(seedX, 0, 1);
 randomY    = $rdist_normal(seedY, 0, 1);
 randomZ    = $rdist_normal(seedZ, 0, 1);

 sigmaX     = randomX/mag(randomX, randomY, randomZ);
 sigmaY     = randomY/mag(randomX, randomY, randomZ);
 sigmaZ     = randomZ/mag(randomX, randomY, randomZ);

 Heffx_int  = getHeffx(mx_int, sigmaX, V(p,n));
 Heffy_int  = getHeffy(my_int, sigmaY, V(p,n));
 Heffz_int  = getHeffz(mz_int, sigmaZ, V(p,n));

//---------------------------------------------------------------//
// Now use intermediate value in final value computation
//---------------------------------------------------------------//
mxpX    = crossX(mx_int, my_int, mz_int, -1, 0, 0);
mxpY    = crossY(mx_int, my_int, mz_int, -1, 0, 0);
mxpZ    = crossZ(mx_int, my_int, mz_int, -1, 0, 0);

mxHeffx = crossX(mx_int, my_int, mz_int, \
                 Heffx_int, Heffy_int, Heffz_int);
mxHeffy = crossY(mx_int, my_int, mz_int, \
                 Heffx_int, Heffy_int, Heffz_int);
mxHeffz = crossZ(mx_int, my_int, mz_int, \
                 Heffx_int, Heffy_int, Heffz_int);

```

```
dmx           = LLGx(mx_int, my_int, mz_int, Heffx_int, Heffy_int, \
                       Heffz_int, mxHeffx, mxHeffy, mxHeffz, \
           mxpX, mxpY, mxpZ, J);
dmy           = LLGy(mx_int, my_int, mz_int, Heffx_int, Heffy_int, \
                       Heffz_int, mxHeffx, mxHeffy, mxHeffz, \
           mxpX, mxpY, mxpZ, J);
dmz           = LLGz(mx_int, my_int, mz_int, Heffx_int, Heffy_int, \
                       Heffz_int, mxHeffx, mxHeffy, mxHeffz, \
           mxpX, mxpY, mxpZ, J);

Mx            = mx_old + (t_step/2)*(dmx + dmx_int);
My            = my_old + (t_step/2)*(dmy + dmy_int);
Mz            = mz_old + (t_step/2)*(dmz + dmz_int);

mx            = Mx/mag(Mx, My, Mz);
my            = My/mag(Mx, My, Mz);
mz            = Mz/mag(Mx, My, Mz);

Heffx         = getHeffx(mx, sigmaX, V(p,n));
Heffy         = getHeffy(my, sigmaY, V(p,n));
Heffz         = getHeffz(mz, sigmaZ, V(p,n));

//---------------------------------------------------------------//
// Update the resistance of the MTJ
//---------------------------------------------------------------//
R             = getRes(mx);

end
endmodule
```

Algorithm A.1: MTJ Verilog-A Model Code.

References

[1] S. Williams *et al.*, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *ACM/IEEE Conf. Supercomputing (SC'07)*, November 2007, pp. 1–12. 2, 11, 33, 72

[2] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *ACM/IEEE Conf. Supercomputing (SC'09)*, November 2009, pp. 1–11. 2, 11, 33, 72

[3] F. Ren, R. Dorrance, W. Xu, and D. Marković, "A Single-Precision Compressive Sensing Signal Reconstruction Engine on FPGAs," in *2013 23rd International Conference on Field Programmable Logic and Applications (FPL'13)*, September 2013, pp. 1–4. 2

[4] S. A. Wolf *et al.*, "The Promise of Nanomagnetics and Spintronics for Future Logic and Universal Memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2155 –2168, December 2010. 3, 4, 90, 91, 92, 94, 101

[5] B. F. Cockburn, "The Emergence of High-Density Semiconductor-Compatible Spintronic Memory," in *International Conference on MEMS, NANO and Smart Systems, 2003. Proceedings.*, July 2003, pp. 321–326. 3, 105

[6] S. Tehrani *et al.*, "Magnetoresistive random access memory using magnetic tunnel junctions," *Proceedings of the IEEE*, vol. 91, no. 5, pp. 703–714, May 2003. 3, 101

[7] M. E. Flatte, "Spintronics," *IEEE Transactions on Electron Devices*, vol. 54, no. 5, pp. 907–920, May 2007. 4, 91, 92, 94

[8] J. Z. Sun, "Spin Angular Momentum Transfer in Current-Perpendicular Nanomagnetic Junctions," *IBM Journal of Research and Development*, vol. 50, no. 1, pp. 81–100, January 2006. 5, 94, 96

[9] J. C. Slonczewski, "Current-Driven Excitation of Magnetic Multilayers," *Journal of Magnetism and Magnetic Materials*, vol. 159, no. 1-2, pp. L1–L7, 1996. 5, 95, 97, 103, 105

[10] E. Chen *et al.*, "Advances and Future Prospects of Spin-Transfer Torque Random Access Memory," *Magnetics, IEEE Transactions on*, vol. 46, no. 6, pp. 1873–1878, June 2010. 5, 103

[11] M. Hosomi *et al.*, "A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram," in *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, December 2005, pp. 459–462. 5, 103

[12] R. Dorrance, F. Ren, and D. Marković, "A Scalable Sparse Matrix-vector Multiplication Kernel for Energy-efficient Sparse-BLAS on FPGAs," in *ACM/SIGDA Int. Symp. Field-programmable Gate Arrays (FPGA'14)*, Feburary 2014, pp. 161–170. 5, 45, 50

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Trans. Math. Software*, vol. 5, no. 3, pp. 308–323, September 1979. 8

[14] S. Sun, M. Monga, P. Jones, and J. Zambreno, "An i/o bandwidth-sensitive sparse matrix-vector multiplication engine on fpgas," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 59, no. 1, pp. 113–123, January 2012. 8, 11, 45, 50, 51, 58

[15] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a universal fpga matrix-vector multiplication architecture," in *Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, April 2012, pp. 9–16. 8, 11, 45, 50, 58

[16] G. Goumas *et al.*, "Understanding the Performance of Sparse Matrix-Vector Multiplication," in *Parallel Distrib. Network-Based Processing (PDP 2008)*, February 2008, pp. 283–292. 8, 11, 32, 45, 50, 58

[17] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," in *Comput. Networks ISDN*, 1998, pp. 107–117. 11

[18] J.-J. Fernandez, "Computational methods for electron tomography," *Micron*, vol. 43, no. 10, pp. 1010–1030, 2012. 11

[19] A. Bustamam, K. Burrage, and N. A. Hamilton, "Fast Parallel Markov Clustering in Bioinformatics Using Massively Parallel Computing on GPU with CUDA and ELLPACK-R Sparse Format," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 9, no. 3, pp. 679–692, May 2012. 11, 20, 22

[20] S. M. van Dongen, "Graph Clustering by Flow Simulation," Ph.D. dissertation, University of Utrecht, The Netherlands, 2000. 11

[21] S. Van Dongen, "Graph Clustering Via a Discrete Uncoupling Process," *SIAM J. Matrix Anal. Appl.*, vol. 30, no. 1, pp. 121–141, February 2008. 11, 20, 22

[22] T. Harlow, J. P. Gogarten, and M. Ragan, "A hybrid clustering approach to recognition of protein families in 114 microbial genomes," *BMC Bioinformatics*, vol. 5, no. 1, p. 45, 2004. 11, 20

[23] S. Wong and M. A. Ragan, "MACHOS: Markov clusters of homologous subsequences," *Bioinformatics*, vol. 24, no. 13, pp. i77–i85, 2008. 11, 20

[24] S. Brohee and J. van Helden, "Evaluation of clustering algorithms for protein-protein interaction networks," *BMC Bioinformatics*, vol. 7, no. 1, p. 488, 2006. 11, 20

[25] J. Sun, G. Peterson, and O. Storaasli, "Mapping sparse matrix-vector multiplication on fpgas," in *Reconfigurable Systems Summer Institute (RSSI 2007)*, July 2007, pp. 1–10. 11, 45, 50, 58

[26] "Intel Math Kernel library." http://software.intel.com/en-us/intel-mkl. 11, 12, 76

[27] "Nvidia cuBLAS." http://developer.nvidia.com/cublas. 11, 12

[28] "Nvidia cuSPARSE." http://developer.nvidia.com/cusparse. 11, 12, 76

[29] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface (3rd ed.).* Morgan Kaufmann Publishers Inc., 2013. 13

[30] A. Fog, "4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs." http://www.agner.org/optimize/. 15

[31] P. Gepner, D. L. Fraser, and V. Gamayunov, "Evaluation of the 3rd generation Intel Core Processor focusing on HPC applications," in *Int. Conf. Parallel Distrib. Process. Techn. Applicat. (PDPTA 2012)*, July 2009, pp. 818–823. 16

[32] "Introducing the GeForce GTX TITAN." http://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-titan. 18

[33] B. Schwikowski, P. Uetz, and S. Fields, "A network of protein?protein interactions in yeast," *Nat. Biotech.*, vol. 18, pp. 1257–1261, December 2000. 24

[34] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," 1999. 23, 25

[35] G. Iván and V. Grolmusz, "When the Web meets the cell: using personalized PageRank for analyzing protein interaction networks," *Bioinformatics*, vol. 27, no. 3, pp. 405–407, 2011. 23

[36] K. Voevodski, S.-H. Teng, and Y. Xia, "Finding local communities in protein networks," *BMC Bioinformatics*, vol. 10, no. 1, p. 297, 2009. 23

[37] E. J. Candès, "Accelerating Phylogeny-Aware Short DNA Read Alignment with FPGAs," in *Proc. Int. Congr. Mathematicians (FCCM'11)*, vol. 3, 2006, pp. 1433–1452. 25, 26

[38] W. Dai, M. A. Sheikh, O. Milenkovic, and R. G. Baraniuk, "Compressive Sensing DNA Microarrays," *EURASIP J Bioinform Syst Biol*, vol. 2009, no. 1, p. 162824, 2009. 26

[39] X. Yan and F. Sun, "Testing gene set enrichment for subset of genes: Sub-GSE," *BMC Bioinformatics*, vol. 9, no. 1, p. 362, 2008. 26

[40] A. Schliep, D. Torney, and S. Rahmann, "Group Testing with DNA Chips: Generating Designs and Decoding Experiments," in *Proc. IEEE Bioinformatics Conf. (CSB 2003)*, August 2003, pp. 84–91. 26

[41] H. Tan *et al.*, "A Computational model for compressed sensing RNAi cellular screening," *BMC Bioinformatics*, vol. 13, no. 1, p. 337, 2012. 26

[42] J. J. Goeman, S. A. van de Geer, F. de Kort, and H. C. van Houwelingen, "A global test for groups of genes: testing association with a clinical outcome," *Bioinformatics*, vol. 20, no. 1, pp. 93–99, 2004. 26

[43] A. Emad and O. Milenkovic, "CaSPIAN: A Causal Compressive Sensing Algorithm for Discovering Directed Interactions in Gene Networks," *PLoS One*, vol. 9, no. 3, 2014. 26

[44] A. Gilbert and P. Indyk, "Sparse Recovery Using Sparse Matrices," *Proc. IEEE*, vol. 98, no. 6, pp. 937–947, June 2010. 26

[45] J.-J. Fernández *et al.*, "High-performance electron tomography of complex biological specimens," *J. Struct. Biol.*, vol. 138, no. 12, pp. 6–20, 2002. 26

[46] C. Sorzano *et al.*, "Marker-free image registration of electron tomography tilt-series," *BMC Bioinformatics*, vol. 10, no. 1, p. 124, 2009. 26

[47] X. Wan, F. Zhang, Q. Chu, and Z. Liu, "High-performance blob-based iterative three-dimensional reconstruction in electron tomography using multi-GPUs," *BMC Bioinformatics*, vol. 13, no. Suppl 10, p. S4, 2012. 26

[48] P. Gilbert, "Iterative methods for the three-dimensional reconstruction of an object from projections," *J Theor Biol.*, vol. 36, no. 1, pp. 105–117, 1972. 26, 27

[49] B. Subramaniam, W. Saunders, T. Scogland, and W. chun Feng, "Trends in energy-efficient computing: A perspective from the Green500," in *Int. Green Comp. Conf. (IGCC'13)*, June 2013, pp. 1–8. 30

[50] "TOP500." http://www.top500.org. 30

[51] S. Gilani, N. S. Kim, and M. Schulte, "Energy-efficient floating-point arithmetic for digital signal processors," in *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, November 2011, pp. 1823–1827. 31

[52] K. Lenzi and O. Saotome, "Optimized math functions for a fixed-point dsp architecture," in *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*, October 2007, pp. 125–132. 31

[53] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *Computers, IEEE Transactions on*, vol. 60, no. 7, pp. 913–922, July 2011. 31

[54] I. Manousakis and D. Nikolopoulos, "BTL: A Framework for Measuring and Modeling Energy in Memory Hierarchies," in *IEEE Int. Symp. Computer Architecture and High Performance Computing (SBAC-PAD)*, October 2012, pp. 139–146. 31, 32

[55] B. Flipsen *et al.*, "Environmental sizing of smartphone batteries," in *Electronics Goes Green 2012+ (EGG), 2012*, September 2012, pp. 1–9. 31

[56] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, August 2008. 41, 65

[57] G. Kuzmanov and M. Taouil, "Reconfigurable sparse/dense matrix-vector multiplier," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, December 2009, pp. 483–488. 45, 50

[58] Y. Zhang *et al.*, "Fpga vs. gpu for sparse matrix vector multiply," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, December 2009, pp. 255–262. 45, 50, 51, 58

[59] C. Lin, H. Kwok-Hay So, and P. Leong, "A model for matrix multiplication performance on fpgas," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, September 2011, pp. 305–310. 45, 50

[60] T. Vanevenhoven, "High-Level Implementation of Bit-and Cycle-Accurate Floating-Point DSP Algorithms with Xilinx FPGAs," 2011. 45

[61] S. Choi, R. Scrofano, V. K. Prasanna, and J.-W. Jang, "Energy-efficient Signal Processing Using FPGAs," in *ACM/SIGDA Int. Symp. Field-programmable Gate Arrays (FPGA'03)*, 2003, pp. 225–234. 45

[62] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-window Applications," in *ACM/SIGDA Int. Symp. Field-programmable Gate Arrays (FPGA'03)*, 2003, pp. 47–56. 45

[63] I. Kuon, R. Tessier, and J. Rose, "FPGA Architecture: Survey and Challenges," *Found. Trends Electron. Des. Autom.*, vol. 2, pp. 135–253, February 2008. 45, 46

[64] B. Calhoun *et al.*, "Flexible Circuits and Architectures for Ultralow Power," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 267–282, February 2010. 45

[65] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 2, pp. 203–215, February 2007. 45

[66] W.-G. Wang *et al.*, "Rapid thermal annealing study of magnetoresistance and perpendicular anisotropy in magnetic tunnel junctions based on mgo and cofeb," *Applied Physics Letters*, vol. 99, no. 10, pp. –, 2011. 46, 103

[67] V. Konda, "Fully Connected Generalized Butterfly Fat Tree Networks," U.S. Patent 0 172 349, July 8, 2010. 46

[68] F. Li, Y. Lin, and L. He, "Vdd Programmability to Reduce FPGA Interconnect Power," in *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, November 2004, pp. 760–765. 46

[69] Y. Lin, F. Li, and L. He, "Routing Track Duplication with Fine-Grained Power-Gating for FPGA Interconnect Power Reduction," in *Asia and South Pacific Design Automation Conference, 2005.*, vol. 1, January 2005, pp. 645–650. 46

[70] Y. Hu, Y. Lin, L. He, and T. Tuan, "Physical Synthesis for FPGA Interconnect Power Reduction by dual-Vdd Budgeting and Retiming," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, pp. 30:1–30:29, April 2008. 46

[71] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *ACM/SIGDA Int. Symp. Field-programmable Gate Arrays (FPGA'05)*, Feburary 2005, pp. 63–74. 50, 51, 52, 58

[72] D. Gregg *et al.*, "Fpga based sparse matrix vector multiplication using commodity dram memory," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, August 2007, pp. 786–791. 50, 52, 58

[73] "ROACH." https://casper.berkeley.edu/wiki/ROACH. 52, 53, 54, 57, 60, 68, 76

[74] "Collaboration for Astronomy Signal Processing and Engineering Research (CASPER)." https://casper.berkeley.edu. 52

[75] "Virtex-5 Family Overview."," http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf. 52

[76] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, December 2011. 72, 74

[77] H. Jeong, S. P. Mason, A.-L. Barabasi, and Z. N. Oltvai, "Lethality and centrality in protein networks," *Nature*, vol. 411, pp. 41–42, May 2001. 74

[78] D. Bu *et al.*, "Topological structure analysis of the proteinprotein interaction network in budding yeast," *Nucl. Acids Res.*, vol. 31, no. 9, pp. 2443–2450, 2003. 74

[79] V. Belcastro *et al.*, "Transcriptional gene network inference from a massive dataset elucidates transcriptome organization and gene function," *Nucl. Acids Res*, vol. 39, no. 20, pp. 8677–8688, 2011. 74

[80] "NVIDIA GeForce GTX 680: The fastest, most efficient GPU ever built." http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf. 79

[81] "Intel Core? i7-4770 Processor." http://ark.intel.com/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3_90-GHz. 79

[82] I. Žutić, J. Fabian, and S. Das Sarma, "Spintronics: Fundamentals and Applications," *Rev. Mod. Phys.*, vol. 76, no. 2, pp. 323–410, April 2004. 90, 93, 94, 95

[83] Y. C. Tao and J. G. Hu, "Superconducting Spintronics: Spin-Polarized Transport in Superconducting Junctions with Ferromagnetic Semiconducting Contact," *Journal of Applied Physics*, vol. 107, no. 4, 2010. 90, 92

[84] M. Jullière, "Tunneling Between Ferromagnetic Films," *Physics Letters A*, vol. 54, no. 3, pp. 225–226, 1975. 90, 97

[85] S. J. Pearton *et al.*, "Spintronics Device Concepts," *Circuits, Devices and Systems, IEE Proceedings*, vol. 152, no. 4, pp. 312–322, August 2005. 91, 93

[86] J. Mathon and A. Umerski, "Theory of Runneling Magnetoresistance of an Epitaxial Fe/MgO/Fe(001) Junction," *Phys. Rev. B*, vol. 63, no. 22, p. 220403, May 2001. 91

[87] S. Das Sarma, J. Fabian, X. Hu, and I. Zutic, "Issues, Concepts, and Challenges in Spintronics," in *IEEE 58th DRC (Device Research Conference) Conference Digest*, June 2000, pp. 95–98. 91

[88] S. Kos *et al.*, "Modeling Spin-Polarized Electron Transport in Semiconductors for Spintronics Applications," *Computing in Science Engineering*, vol. 9, no. 5, pp. 46–52, September-October 2007. 93

[89] M. E. Flatté, Z. G. Yu, E. Johnston-Halperin, and D. D. Awschalom, "Theory of Semiconductor Magnetic Bipolar Transistors," *Applied Physics Letters*, vol. 82, no. 26, pp. 4740–4742, 2003. 93

[90] S. Chakrabarti *et al.*, "Spin-Polarized Light-Emitting Diodes with Mn-Doped InAs Quantum Dot Nanomagnets as a Spin Aligner," *Nano Letters*, vol. 5, no. 2, pp. 209–212, 2005. 93

[91] P. Bruno, Y. Suzuki, and C. Chappert, "Magneto-optical Kerr Effect in a Paramagnetic Overlayer on a Ferromagnetic Substrate: A Spin-Polarized Quantum Size Effect," *Phys. Rev. B*, vol. 53, no. 14, pp. 9214–9220, April 1996. 93

[92] Z. Diao *et al.*, "Spin Transfer Switching in Dual MgO Magnetic Tunnel Junctions," *Applied Physics Letters*, vol. 90, no. 13, p. 132508, 2007. 95

[93] W. Zhu, H. Li, Y. Chen, and X. Wang, "Current Switching in MgO-Based Magnetic Tunneling Junctions," *Magnetics, IEEE Transactions on*, vol. 47, no. 1, pp. 156–160, January 2011. 95, 124

[94] K. Lee and S. Kang, "Control of Switching Current Asymmetry by Magnetostatic Field in MgO-Based Magnetic Tunnel Junctions," *Electron Device Letters, IEEE*, vol. 30, no. 12, pp. 1353–1355, December 2009. 95

[95] X. Yao, H. Meng, Y. Zhang, and J.-P. Wang, "Improved Current Switching Symmetry of Magnetic Tunneling Junction and Giant Magnetoresistance Devices with Nano-Current-Channel Structure," *Journal of Applied Physics*, vol. 103, no. 7, p. 07A717, 2008. 95

[96] R. H. Koch, J. A. Katine, and J. Z. Sun, "Time-Resolved Reversal of Spin-Transfer Switching in a Nanomagnet," *Phys. Rev. Lett.*, vol. 92, no. 8, p. 088302, February 2004. 96, 97

[97] H. W. Schumacher *et al.*, "Precessional Switching of the Magnetization in Microscopic Magnetic Tunnel Junctions (Invited)," *Journal of Applied Physics*, vol. 93, no. 10, pp. 7290–7294, 2003. 96

[98] T. L. Gilbert, "A Phenomenological Theory of Damping in Ferromagnetic Materials," *IEEE Transactions on Magnetics*, vol. 40, no. 6, pp. 3443–3449, November 2004. 97

[99] L. Landau and E. Lifshitz, "On the Theory of the Dispersion of Magnetic Permeability in Ferromagnetic Bodies," *Physikalische zeitschrift der Sowjetunion*, vol. 8, pp. 153–169, 1935. 97

[100] T. Moriyama *et al.*, "Tunnel Magnetoresistance and Spin Torque Switching in MgO-based Magnetic Tunnel Junctions with a Co/Ni Multilayer Electrode," *Applied Physics Letters*, vol. 97, no. 7, p. 072513, 2010. 97, 105, 106, 114

[101] Z. M. Zeng *et al.*, "Effect of resistance-area product on spin-transfer switching in mgo-based magnetic tunnel junction memory cells," *Applied Physics Letters*, vol. 98, no. 7, p. 072512, 2011. 98

[102] Y. Higo *et al.*, "Thermal Activation Effect on Spin Transfer Switching in Magnetic Tunnel Junctions," *Applied Physics Letters*, vol. 87, no. 8, p. 082502, 2005. 97

[103] M. Pakala *et al.*, "Critical Current Distribution in Spin-Transfer-Switched Magnetic Tunnel Junctions," *Journal of Applied Physics*, vol. 98, no. 5, p. 056107, 2005. 97

[104] T. Aoki, Y. Ando, M. Oogane, and H. Naganuma, "Reproducible Trajectory on Sub-nanosecond Spin-Torque Magnetization Switching Under a Zero-Bias Field for MgO-Based Ferromagnetic Tunnel Junctions," *Applied Physics Letters*, vol. 96, no. 14, p. 142502, 2010. 97

[105] E. R. Nowak, M. B. Weissman, and S. S. P. Parkin, "Electrical Noise in Hysteretic Ferromagnet–Insulator–Ferromagnet Tunnel Junctions," *Applied Physics Letters*, vol. 74, no. 4, pp. 600–602, 1999. 97, 100, 101

[106] S. X. Huang, T. Y. Chen, and C. L. Chien, "Spin Polarization of Amorphous CoFeB Determined by Point-Contact Andreev Reflection," *Applied Physics Letters*, vol. 92, no. 24, p. 242509, 2008. 97

[107] J. S. Moodera and G. Mathon, "Spin Polarized Tunneling in Ferromagnetic Junctions," *Journal of Magnetism and Magnetic Materials*, vol. 200, no. 1-3, pp. 248 – 273, 1999. 97, 99

[108] Y. Lu *et al.*, "Bias Voltage and Temperature Dependence of Magnetotunneling Effect," *Journal of Applied Physics*, vol. 83, no. 11, pp. 6515–6517, 1998. 97, 99, 112

[109] X. Liu *et al.*, "Thermal Stability of Magnetic Tunneling Junctions with MgO Barriers for High Temperature Spintronics," *Applied Physics Letters*, vol. 89, no. 2, p. 023504, 2006. 97

[110] C. H. Shang, J. Nowak, R. Jansen, and J. S. Moodera, "Temperature Dependence of Magnetoresistance and Surface Magnetization in Ferromagnetic Tunnel Junctions," *Phys. Rev. B*, vol. 58, no. 6, pp. R2917–R2920, August 1998. 99, 111, 117

[111] J. G. Simmons, "Generalized Formula for the Electric Tunnel Effect between Similar Electrodes Separated by a Thin Insulating Film," *Journal of Applied Physics*, vol. 34, no. 6, pp. 1793–1803, 1963. 99

[112] T. Zhu, X. Xiang, and J. Q. Xiao, "Bias Dependence of Tunneling Magnetoresistance on Ferromagnetic Electrode Thickness," *Applied Physics Letters*, vol. 82, no. 16, pp. 2676–2678, 2003. 99

[113] G. G. Cabrera and N. García, "Low Voltage I–V Characteristics in Magnetic Tunneling Junctions," *Applied Physics Letters*, vol. 80, no. 10, pp. 1782–1784, 2002. 99

[114] S. Chatterjee, S. Salahuddin, S. Kumar, and S. Mukhopadhyay, "Analysis of Thermal Behaviors of Spin-Torque-Transfer RAM: A Simulation Study," in *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, August 2010, pp. 13–18. 99, 100

[115] Y. Zhang *et al.*, "Micromagnetic Study of Hotspot and Thermal Effects on Spin-Transfer Switching in Magnetic Tunnel Junctions," *Journal of Applied Physics*, vol. 101, no. 10, p. 103905, 2007. 99

[116] S. Chaudhuri *et al.*, "Design of TAS-MRAM Prototype for NV Embedded Memory Applications," in *Memory Workshop (IMW), 2010 IEEE International*, May 2010, pp. 1 –4. 100

[117] J. Z. Sun *et al.*, "High-Bias Backhopping in Nanosecond Time-Domain Spin-Torque Switches of MgO-based Magnetic Tunnel Junctions," *Journal of Applied Physics*, vol. 105, no. 7, p. 07D109, 2009. 100

[118] T. Min *et al.*, "Back-Hopping after Spin Torque Transfer Induced Magnetization Switching in Magnetic Tunneling Junction Cells," *Journal of Applied Physics*, vol. 105, no. 7, p. 07D126, 2009. 100

[119] S. Ingvarsson *et al.*, "Low-Frequency Magnetic Noise in Micron-Scale Magnetic Tunnel Junctions," *Phys. Rev. Lett.*, vol. 85, no. 15, pp. 3289–3292, Oct 2000. 100, 101

[120] K. B. Klaassen, J. C. L. van Peppen, and X. Xing, "Noise in Magnetic Tunnel Junction Devices," *Journal of Applied Physics*, vol. 93, no. 10, pp. 8573–8575, 2003. 100

[121] K. Shimazawa *et al.*, "Frequency Response of Common Lead and Shield Type Magnetic Tunneling Junction Head," *Magnetics, IEEE Transactions on*, vol. 37, no. 4, pp. 1684–1686, July 2001. 100

[122] A. F. M. Nor *et al.*, "Low-Frequency Noise in MgO Magnetic Tunnel Junctions," *Journal of Applied Physics*, vol. 99, no. 8, p. 08T306, 2006. 100, 101

[123] K. Itoh, T. Watanabe, S. Kimura, and T. Sakata, "Reviews and Prospects of High-Density RAM Technology," in *Semiconductor Conference, 2000. CAS 2000 Proceedings. International*, vol. 1, 2000, pp. 13–22. 101

[124] M. Durlam *et al.*, "Nonvolatile RAM Based on Magnetic Tunnel Junction Elements," in *Solid-State Circuits Conference, 2000. Digest of Technical Papers. ISSCC. 2000 IEEE International*, February 2000, pp. 130–131. 101

[125] I. Prejbeanu *et al.*, "Thermally Assisted Switching in Exchange-biased Storage Layer Magnetic Tunnel Junctions," *Magnetics, IEEE Transactions on*, vol. 40, no. 4, pp. 2625–2627, July 2004. 102

[126] R. Sinclair and A. Pohm, "Scaling and Power of Thermally Written MRAM," in *Non-Volatile Memory Technology Symposium, 2004*, November 2004, pp. 110–117. 102

[127] J. Deak, J. Daughton, and A. Pohm, "Effect of Resistance-Area-Product and Thermal Environment on Writing of Magneto-Thermal MRAM," *Magnetics, IEEE Transactions on*, vol. 42, no. 10, pp. 2721–2723, October 2006. 102

[128] P. K. Amiri and K. L. Wang, "Voltage-controlled magnetic anisotropy in spintronic device," *SPIN*, vol. 2, no. 3, p. 1240002, 2012. 103

[129] S. Ikeda *et al.*, "A perpendicular-anisotropy cofeb-mgo magnetic tunnel junction," *Nat Mater*, vol. 9, no. 9, pp. 721–724, 2010. 103

[130] J. Zhu *et al.*, "Voltage-Induced Ferromagnetic Resonance in Magnetic Tunnel Junctions," *Phys. Rev. Lett.*, vol. 108, p. 197203, May 2012. 104, 108, 142

[131] T. Nozaki *et al.*, "Voltage-induced perpendicular magnetic anisotropy change in magnetic tunnel junctions," *Applied Physics Letters*, vol. 96, no. 2, pp. –, 2010. 104

[132] T. Maruyama *et al.*, "Large voltage-induced magnetic anisotropy change in a few atomic layers of iron," *Nat. Nano.*, vol. 4, pp. 158–161, March 2009. 104, 109

[133] J. G. Alzate *et al.*, "Voltage-Induced Switching of Nanoscale Magnetic Tunnel Junctions," in *Proceedings of the International Electron Devices Meeting (IEDM'12)*, December 2012, pp. 29.5.1–29.5.4. 104, 108, 109, 110, 139, 142

[134] Y. Shiota *et al.*, "Induction of coherent magnetization switching in a few atomic layers of feco using voltage pulses," *Nat. Mater.*, vol. 11, pp. 39–43, January 2012. 104

[135] W.-G. Wang, M. Li, S. Hageman, and C. L. Chien, "Electric-field-assisted switching in magnetic tunnel junctions," *Nat. Mater.*, vol. 11, pp. 64–68, January 2012. 104

[136] Y. Huai, "Spin-Transfer Torque MRAM (STT-MRAM): Challenges and Prospects," *AAPPS Bulletin*, vol. 18, no. 6, pp. 33–40, December 2008. 105

[137] M. R. Scheinfein, "LLG Micromagnetics Simulator." [Online]. Available: http://llgmicro.home.mindspring.com 105, 118

[138] J. Z. Sun, "Spin-Current Interaction with a Monodomain Magnetic Body: A Model Study," *Phys. Rev. B*, vol. 62, no. 1, pp. 570–578, July 2000. 107, 115

[139] J. A. Osborn, "Demagnetizing Factors of the General Ellipsoid," *Phys. Rev.*, vol. 67, no. 11-12, pp. 351–357, June 1945. 108, 115

[140] W. G. Wang and C. L. Chien, "Voltage-induced switching in magnetic tunnel junctions with perpendicular magnetic anisotropy," *J. Phys. D*, vol. 46, no. 7, p. 074004, 2013. 109, 110

[141] P. K. Amiri, R. Dorrance, D. Marković, and K. L. Wang, "Nonvolatile Magneto-Electric Random Access Memory Circuit with Burst Writing and Back-to-Back Reads," U.S. Patent 8 988 923, March 24, 2015. 110, 139, 147

[142] P. K. Amiri, R. Dorrance, D. Marković, and K. L. Wang, "Read-Disturbance-Free Nonvolatile Content Addressable Memory (CAM)," U.S. Patent 9 047 950, June 2, 2015. 110

[143] P. Weiss, "L'hypothèse du Champ Moléculaire et la Propriété Ferromagnétique," *J. Phys. Theor. Appl.*, vol. 6, no. 1, pp. 661–690, 1907. 110

[144] A. Raghunathan, Y. Melikhov, J. E. Snyder, and D. C. Jiles, "Modeling the Temperature Dependence of Hysteresis Based on Jiles-Atherton Theory," *IEEE Transactions on Magnetics*, vol. 45, no. 10, pp. 3954–3957, October 2009. 111

[145] S. R. Min *et al.*, "Etch Characteristics of Magnetic Tunnel Junction Stack with Nanometer-Sized Patterns for Magnetic Random Access Memory," *Thin Solid Films, Proceedings of the International Symposium on Dry Process, 2006. (DPS 2006).*, vol. 516, no. 11, pp. 3507–3511, November 2008. 113

[146] R. Beach *et al.*, "A Statistical Study of Magnetic Tunnel Junctions for High-Density Spin Torque Transfer-MRAM (STT-MRAM)," in *Electron Devices Meeting, 2008. IEDM 2008. IEEE International*, December 2008, pp. 1–4. 113

[147] H. Chang and J. Burns, "Demagnetizing and Stray Fields of Elliptical Films," *Journal of Applied Physics*, vol. 37, no. 8, pp. 3240–3245, July 1966. 115

[148] V. Korenivski and R. Leuschner, "Thermally Activated Switching in Nanoscale Magnetic Tunnel Junctions," *IEEE Transactions on Magnetics*, vol. 46, no. 6, pp. 2101–2103, June 2010. 115

[149] J. Sun and D. Ralph, "Magnetoresistance and Spin-Transfer Torque in Magnetic Tunnel Junctions," *Journal of Magnetism and Magnetic Materials*, vol. 320, no. 7, pp. 1227–1237, 2008. 115

[150] X. Kou, J. Schmalhorst, A. Thomas, and G. Reiss, "Temperature Dependence of the Resistance of Magnetic Tunnel Junctions with MgO Barrier," *Applied Physics Letters*, vol. 88, no. 21, p. 212115, 2006. 116, 117

[151] P. Padhan *et al.*, "Frequency-Dependent Magnetoresistance and Magnetocapacitance Properties of Magnetic Tunnel Junctions with MgO Tunnel Barrier," *Applied Physics Letters*, vol. 90, no. 14, p. 142105, 2007. 116

[152] P. Wiśniowski *et al.*, "Temperature Dependence of Tunnel Magnetoresistance and Magnetization of IrMn Based MTJ," *Physica Status Solidi*, vol. 201, pp. 1648–1652, 2004. 117

[153] W. Zhao *et al.*, "New Non-Volatile Logic Based on Spin-MTJ," *physica status solidi (a)*, vol. 205, no. 6, pp. 1373–1377, 2008. 122

[154] X. F. Han and A. C. C. Yu, "Patterned Magnetic Tunnel Junctions with Al Conduction Layers: Fabrication and Reduction of Pinhole Effect," *Journal of Applied Physics*, vol. 95, no. 2, pp. 764–766, 2004. 122

[155] S. Isogami, M. Tsunoda, and M. Takahashi, "30-nm Scale Fabrication of Magnetic Tunnel Junctions using EB Assisted CVD Hard Masks," *Magnetics, IEEE Transactions on*, vol. 41, no. 10, pp. 3607–3609, October 2005. 122

[156] C. J. Lin *et al.*, "45nm Low Power CMOS Logic Compatible Embedded STT MRAM Utilizing a Reverse-Connection 1T/1MTJ Cell," in *Electron Devices Meeting (IEDM), 2009 IEEE International*, December 2009, pp. 1–4. 124, 134

[157] T. Takenaga *et al.*, "Control of Pinned Layer Magnetization Direction in Spin-Valve-Type Magnetic Tunnel Junction with an IrMn Layer," *Journal of Applied Physics*, vol. 95, no. 11, pp. 6795–6797, 2004. 124

[158] H. Park *et al.*, "Analysis of STT-RAM Cell Design with Multiple MJTs Per Access," in *Proceedings of the ACM/IEEE International Symposium on Nanoscale Architectures (NANOARCH'11)*, June 2011, pp. 53–58. 127

[159] R. Dorrance, "Modeling and Design of STT-MRAMs," Master's thesis, University of California, Los Angeles, June 2011. 129

[160] R. Dorrance *et al.*, "Scalability and Design-Space Analysis of a 1T-1MTJ Memory Cell," in *Proceedings of the ACM/IEEE International Symposium on Nanoscale Architectures (NANOARCH'11)*, June 2011, pp. 32–36. 129

[161] R. Dorrance *et al.*, "Scalability and Design-Space Analysis of a 1T-1MTJ Memory Cell for STT-RAMs," *IEEE Transactions on Electron Devices (TED)*, vol. 59, no. 4, pp. 878–887, April 2012. 129

[162] P. Amiri *et al.*, "Low Write-Energy Magnetic Tunnel Junctions for High-Speed Spin-Transfer-Torque MRAM," *Electron Device Letters, IEEE*, vol. 32, no. 1, pp. 57–59, January 2011. 130

[163] P. K. Amiri *et al.*, "Switching Current Reduction Using Perpendicular Anisotropy in CoFeB-MgO Magnetic Tunnel Junctions," *Applied Physics Letters*, vol. 98, no. 11, p. 112507, 2011. 130

[164] G. E. Rowlands *et al.*, "Deep Subnanosecond Spin Torque Switching in Magnetic Tunnel Junctions with Combined In-Plane and Perpendicular Polarizers," *Applied Physics Letters*, vol. 98, no. 10, p. 102509, 2011. 130

[165] R. Nebashi *et al.*, “A 90nm 12ns 32Mb 2T1MTJ MRAM,” in *ISSCC 2009*, February 2009, pp. 462–463, 463a. 134

[166] D. Halupka *et al.*, “Negative-Resistance Read and Write Schemes for STT-MRAM in 0.13 μm CMOS,” in *ISSCC 2010*, February 2010, pp. 256–257. 134

[167] K. Tsuchida *et al.*, “A 64Mb MRAM with Clamped-Reference and Adequate-Reference Schemes,” in *ISSCC 2010*, February 2010, pp. 258–259. 134

[168] F. Ren *et al.*, “A Body-Voltage-Sensing-Based Short Pulse Reading Circuit for Spin-Torque Transfer RAMs (STT-RAMs),” in *Proceedings of 13th International Symposium on Quality Electronic Design (ISQED'12)*, March 2012, pp. 275–282. 137

[169] A. Chen, Z. Krivokapic, and M.-R. Lin, “A Comprehensive Model for Crossbar Memory Arrays,” in *Proc Device Research Conference*, June 2012, pp. 219–220. 138

[170] R. Dorrance *et al.*, “Diode-MTJ Crossbar Memory Cell Using Voltage-Induced Unipolar Switching for High-Density MRAM,” *IEEE Electron Device Letters (EDL)*, vol. 34, no. 6, pp. 753–755, June 2013. 139