

# **Acceleration Techniques of Sparse Linear Algebra on Emerging Architectures**

by

Siying Feng

A dissertation submitted in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy  
(Computer Science and Engineering)  
in the University of Michigan  
2022

Doctoral Committee:

Associate Professor Ronald G. Dreslinski, Chair  
Professor Scott Mahlke  
Professor Trevor N. Mudge  
Professor Zhengya Zhang

Siyang Feng

fengsy@umich.edu

ORCID iD: 0000-0002-2685-4149

© Siyang Feng 2022

*Dedicated to my parents, Jianping Feng and Xianshi Feng,  
for all the love, support, and sacrifice they have given to me.*

## ACKNOWLEDGMENTS

My greatest thank goes to my advisor, Professor Ronald Dreslinski, who introduced me to academic research and gave me invaluable support and guidance throughout my Ph.D. journey. I also express my gratitude to Professor Trevor Mudge, who is always willing to share his wisdom and advise me on my studies. I would like to thank Professor Scott Mahlke and Professor Zhengya Zhang for providing important, wider perspectives to my work and guiding me as my dissertation committee member.

I gratefully acknowledge Professor Chaitali Chakrabarti, David Blaauw, Hun-Seok Kim, Christophe Dubach, Murray Cole, and Michael O'Boyle for their insights and guidance in the Defense Advanced Research Projects Agency (DARPA) Software-Defined Hardware (SDH) program, which supports a significant portion of work in this dissertation. I am also extremely grateful for the fellowships from Rackham Graduate School and the Computer Science and Engineering (CSE) department that enabled me to focus on research and explore my own research interests.

I sincerely thank all my collaborators for their contributions to the works presented in this dissertation. I want to give my special thanks to Subhankar Pal, who led the development of Transmuter and has been my closest collaborator, my best friend, and my mentor. I would also like to thank my other close collaborators: Jiawen Sun, Aporva Amarnath, Kuan-Yu Chen, Dong-hyeon Park, Liu Ke, Yichen Yang, for their valuable efforts and innovative ideas. I additionally thank my lab mates and friends: Tutu Ajayi, Nishil Talati, Javad Bagherzadeh, Heewoo Kim, Austin Rovinski, Jielun Tan, Deepika Natarajan, Morteza Fayazi, Suman Kumar Mallik, Haojie Ye, and Yuhan Chen, for the memorable moments we share that make working in the lab an enjoyable experience.

Last but not least, I want to express my profound gratitude to my family. I would not be who I am today without the love and support of my parents, Jianping Feng and Xianshi Feng. They have always been there for me, encouraging me during tough times, supporting all my decisions, and being my never-ending source of strength. Finally, I would like to thank Xin He, my partner both in work and in life, for keeping me company through all the lows and highs of life.

# TABLE OF CONTENTS

<b>DEDICATION</b> . . . . .	<b>ii</b>
<b>ACKNOWLEDGMENTS</b> . . . . .	<b>iii</b>
<b>LIST OF FIGURES</b> . . . . .	<b>vii</b>
<b>LIST OF TABLES</b> . . . . .	<b>xii</b>
<b>LIST OF PROGRAMS</b> . . . . .	<b>xiii</b>
<b>LIST OF ABBREVIATIONS</b> . . . . .	<b>xiv</b>
<b>ABSTRACT</b> . . . . .	<b>xviii</b>
<b>CHAPTER</b>	
<b>1 Introduction</b> . . . . .	<b>1</b>
1.1 Emerging Architecture Techniques . . . . .	1
1.2 Dissertation Overview . . . . .	4
<b>2 Challenges of Executing Sparse Linear Algebra on Contemporary Hardware</b> .	<b>6</b>
2.1 A Parallelism Analysis on Prominent Desktop Applications . . . . .	6
2.1.1 Methodology . . . . .	7
2.1.2 Evaluation Overview . . . . .	7
2.1.3 Evolution of Concurrency . . . . .	9
2.1.4 Analyses on GPU . . . . .	10
2.1.5 Takeaways . . . . .	13
2.2 Prevalence and Challenges of Sparse Linear Algebra . . . . .	13
<b>3 General-Purpose Acceleration through Reconfigurable Memory Hierarchy</b> . .	<b>17</b>
3.1 Introduction . . . . .	18
3.1.1 Contribution . . . . .	20
3.2 Motivation . . . . .	21
3.2.1 Contemporary Computing Platforms . . . . .	21
3.2.2 Taming the Diversity across Kernels . . . . .	22
3.2.3 Hardware Implication of Disparate Patterns . . . . .	24
3.3 Transmuter Overview . . . . .	26

3.4	Transmuter Architecture Design . . . . .	29
3.4.1	General-purpose Processing Element and Local Control Processor	29
3.4.2	Work and Status Queues . . . . .	29
3.4.3	Reconfigurable Data Cache (R-DCache) . . . . .	30
3.4.4	Reconfigurable Crossbar (R-XBar) . . . . .	30
3.4.5	Synchronization Scratchpad Memory . . . . .	32
3.5	Transmuter Reconfiguration Design . . . . .	32
3.6	Prototype Software Stack . . . . .	34
3.7	Experimental Methodology . . . . .	36
3.7.1	Performance Models . . . . .	36
3.7.2	Power and Area Models . . . . .	38
3.8	Kernel Implementations on Transmuter . . . . .	38
3.8.1	Dense Matrix Multiplication and Convolution . . . . .	39
3.8.2	Fast Fourier Transform . . . . .	41
3.8.3	Sparse Matrix Multiplication . . . . .	42
3.8.4	Performance with Different Configurations . . . . .	45
3.9	Evaluation . . . . .	48
3.9.1	Comparison with the CPU and GPU . . . . .	48
3.9.2	Comparison with FPGA, CGRA, and ASIC . . . . .	50
3.9.3	Power and Area . . . . .	50
3.9.4	End-to-End Workload Analysis . . . . .	50
3.9.5	Throughput and Bandwidth Analysis . . . . .	53
3.9.6	Design Space Exploration . . . . .	54
3.9.7	Control Divergence and Data Reuse Analysis . . . . .	54
3.10	Related Work . . . . .	56
3.11	Conclusion . . . . .	57
<b>4</b>	<b>Intelligent Software and Hardware Reconfiguration for Graph Processing . .</b>	<b>59</b>
4.1	Introduction . . . . .	60
4.2	Background and Related Work . . . . .	62
4.2.1	Graph Frameworks using Software Reconfigurations . . . . .	62
4.2.2	Optimized Hardware Acceleration for Graph Analytics . . . . .	62
4.2.3	Opportunities in Combining Software/Hardware Optimizations . . . . .	63
4.3	CoSPARSE Reconfiguration Layer Design . . . . .	63
4.3.1	Reconfigurable SpMV Implementation . . . . .	64
4.3.2	Workload Balancing Strategies . . . . .	66
4.3.3	Reconfiguration Threshold Analysis . . . . .	67
4.3.4	Graph Analytics Algorithms on CoSPARSE . . . . .	71
4.4	Methodology . . . . .	73
4.5	Evaluation . . . . .	73
4.5.1	Workload Balancing Evaluation . . . . .	74
4.5.2	Comparison against Existing Platforms . . . . .	75
4.6	Conclusion . . . . .	78
<b>5</b>	<b>Near-Memory Multi-way Merge Solution for Sparse Data Merging . . . . .</b>	<b>79</b>

5.1	Introduction . . . . .	80
5.2	Background and Motivation . . . . .	83
5.2.1	Sparse Matrix Formats and Sparse Matrix Transposition . . . . .	83
5.2.2	Characterizations on Sparse Matrix Transposition . . . . .	85
5.3	MeNDA System Architecture . . . . .	87
5.3.1	Algorithm and Dataflow . . . . .	88
5.3.2	Processing Unit (PU) Microarchitecture . . . . .	88
5.3.3	Seamless Back-to-back Merge Sort . . . . .	90
5.3.4	Memory Bandwidth Utilization Optimizations . . . . .	91
5.3.5	Input Operand Co-location and Workload Balancing . . . . .	92
5.3.6	Adaptation to SpMV . . . . .	93
5.4	Programming Model and Interface . . . . .	95
5.4.1	Integrating MeNDA with Existing Platforms . . . . .	96
5.5	Experimental Methodology . . . . .	97
5.5.1	Simulation Methodology . . . . .	98
5.5.2	Baseline and Benchmarks . . . . .	98
5.6	Evaluation . . . . .	99
5.6.1	Comparison with CPU and GPU Baselines . . . . .	99
5.6.2	Area and Power Analysis . . . . .	100
5.6.3	Benefits and Overhead Analysis on End-to-end Workloads . . . . .	101
5.6.4	Memory Bandwidth Utilization Optimization Analysis . . . . .	102
5.6.5	Scalability Analysis . . . . .	103
5.6.6	Matrix Distribution Analysis . . . . .	104
5.6.7	Design Space Exploration . . . . .	104
5.6.8	SpMV Analysis . . . . .	105
5.7	Related Works . . . . .	106
5.8	Conclusion . . . . .	107
<b>6</b>	<b>Conclusion . . . . .</b>	<b>109</b>
	<b>BIBLIOGRAPHY . . . . .</b>	<b>111</b>

## LIST OF FIGURES

1.1	Growing trend in the number of transistors and microprocessor performance and power over the past 42 years [164]. . . . .	2
1.2	Comparison in performance, efficiency, and programmability/flexibility for different architecture designs [129]. . . . .	2
1.3	The growing trend of (a) DNN model complexity over the past 8 years [82] and (b) DRAM capacity, bandwidth, and latency over the past 20 years [143]. . . . .	3
2.1	Comparison between TLP of desktop applications for 2000 [56, 57], 2010 [22], and 2018 [this work]. . . . .	9
2.2	Comparison between GPU utilization of desktop applications for 2010 [22] and 2018 [this work]. . . . .	10
2.3	Transcode rates (GTX 1080 Ti only) and GPU utilization of HandBrake and WinX for 2, 4, and 6 logical cores, showing the effect of GPU offloading. The transcode rates for GTX 680 are not shown as they overlap exactly with those for GTX 1080 Ti. . . . .	11
2.4	GPU utilization of GTX 680 and 1080 Ti for Premiere Pro. Video export with CUDA support shows higher utilization and lower TLP than without CUDA, and the utilization is higher for GTX 680. . . . .	12
2.5	GPU utilization of GTX 680 and 1080 Ti for applications that show substantial use of GPU. VR is excluded as it requires a GPU better than GTX 970. PhoenixMiner does not support GTX 680. . . . .	12
2.6	High-level overview of SpMM using the inner product algorithm [153] . . . . .	15
3.1	<i>Left:</i> Transmuter compared to contemporary platforms in terms of programmability, hardware flexibility, and reconfiguration overhead. <i>Right:</i> Energy-efficiency comparisons for kernels spanning a wide range of arithmetic intensities (FLOPS/B). Note that for ASICs and CGRAs, no single piece of hardware supports all kernels. Transmuter achieves 2.0× better average efficiency over state-of-the-art CGRAs while retaining the programmability of GPPs. . . . .	19
3.2	Trade-offs between programmability and efficiency in prominent computer architectures. . . . .	21



3.3	Fraction of execution time of kernels in applications spanning the domains of ML, signal processing, and graph analytics [207, 139, 93, 43, 130, 120, 44, 131, 35, 27] on a heterogeneous CPU-GPU platform. Some key characteristics, namely arithmetic intensity, data reuse, and divergence, of each kernel are also listed. . . . .	23
3.4	<i>Left:</i> Performance of an SPM over a cache-based single-core system for a synthetic workload with variable access patterns and arithmetic intensity. <i>Right:</i> Performance of a shared cache over a private cache-based 8-core system on a synthetic program with varying access strides and working set overlaps across cores. “Contending” is a case where all cores, in a given cycle, access addresses that map to the same bank in the shared mode. . . . .	24
3.5	High-level Transmuter architecture showing the configurations evaluated in this work, namely a) Trans-SC (L1: shared cache, L2: shared cache), b) Trans-PS (L1: private SPM, L2: private cache), and c, d) Trans-SA (L1: systolic array, L2: private cache). . . . .	27
3.6	a) High-level overview of a host-Transmuter system. b) Transmuter architecture showing 4 tiles and 4 L2 R-DCache banks, along with L2 R-XBars, the synchronization SPM, and interface to off-chip memory. Some L2 R-XBar input connections are omitted for clarity. c) View of a single tile, showing 4 GPEs and the work/status queues interface. Arbiters, instruction paths, and caches (ICaches) are not shown. d) Microarchitecture of an R-XBar, with the circled numbers indicating the mode of operation: ①: ARBITRATE, ②: TRANSPARENT, ③: ROTATE. . . . .	28
3.7	a) Logical view of an R-DCache bank in FIFO+SPM mode, with 4 FIFO partitions, one for each direction in 2D. b) Loads and stores to special addresses corresponding to each direction are mapped to POP and PUSH calls, respectively, into the FIFOs. . . . .	30
3.8	a) Physical and b) logical views of 1D systolic array connections within a Transmuter tile. Spatial dataflow is achieved by the R-XBar rotating between the two port-connection patterns. . . . .	31
3.9	Transmuter software stack. Application code is written using Python and invokes library code for the host, LCPs, and GPEs. The implementations are written by experts using our C++ intrinsics library. Also shown is an example of a correlation kernel on Trans-SA (host library code not shown). The end-user writes standard NumPy code and changes only the import package to <code>transpy.numpy</code> (App:L1). Upon a library call (App:L5), the host performs data transfers and starts execution on Transmuter. The LCP broadcasts the vector $x$ to all GPEs (LCP:L7). Each GPE pops the value (GPE:L4), performs a MAC using its filter value ( $f$ ) and east neighbor’s partial sum (GPE:L7), and sends its partial sum westward (GPE:L11). The last GPE stores the result into HBM. The host returns control to the application after copying back the result vector $y$ . . . . .	34
3.10	Illustration of dense matrix-matrix and matrix-vector multiplication kernels mapped onto Transmuter. . . . .	39

3.11	Top: Mapping of FFT stages onto GPEs in Transmuter (Trans-SA). Bottom: Each GPE executes butterfly operations greedily, leading to a fully-pipelined schedule. . . . .	41
3.12	SpMM mapping on Transmuter. Left: Multiply phase: each GPE multiplies an element of a column of $A$ with a row of $B$ , generating a partial product matrix. Right: Merge phase: each GPE independently streams in partial product matrix rows, performs mergesort, and stores the result. . . . .	42
3.13	Performance of $2 \times 8$ Trans-SC, Trans-PS, and Trans-SA configurations across different inputs for the kernels in Sec. 3.8. All matrix operations are performed on square matrices without loss of generality. Convolution uses $3 \times 3$ filters, 2 input/output channels, and a batch size of 2. . . . .	46
3.14	Cycle breakdown for the kernels in Sec. 3.8. * (red) indicates the best-performing configuration. “Other” comprises stalls due to synchronization and bank conflicts. $\blacktriangledown$ : work imbalance across GPEs ( $\sigma/\mu$ of # FLOPS). Inputs are: 1k (GEMM), 8k (GEMV), 2k (Convolution), 16k (FFT), 4096, 0.64% (SpMM), 4k, 2.6%, dense vector (SpMV). . . . .	47
3.15	Throughput (left) and energy-efficiency (right) improvements of Transmuter over the CPU and GPU. Data is averaged across the inputs: 256-1k (GEMM), 2k-8k (GEMV), 512-2k (Convolution), 4k-16k (FFT), 1k-4k, 0.64% (SpMM), and 2k-4k, 2.6% ( $r_M$ ), 10.2%-100% ( $r_v$ ) (SpMV). Geometric mean improvements for the compute-bound and memory-bound kernels are shown separately. . . . .	48
3.16	Mapping of a multi-kernel, mixed data application, Sinkhorn, on Transmuter. Computation iterates between M-GEMM and DMSpM, with Trans-SC $\leftrightarrow$ Trans-PS reconfiguration before and after DMSpM-Merge. DMSpM-Merge benefits from the private SPMs in Trans-PS, since each GPE works on multiple disjoint lists. . . . .	52
3.17	Per inner-loop iteration energy (left) and EDP (right) comparing Trans-SC, Trans-PS and Reconf. (Trans-SC $\leftrightarrow$ Trans-PS) for Sinkhorn normalized to CPU. Input matrix dimensions and densities are — <i>query</i> : $(8k \times 1)$ , 1%, <i>data</i> : $(8k \times 1k)$ , 1%, <i>M</i> : $(8k \times 8k)$ , 99%. . . . .	53
3.18	Effect of scaling tiles and GPEs per tile on performance and memory bandwidth for GEMM (Trans-SC), GEMV (Trans-SC) and SpMM (Trans-PS). Inputs are: 1k (GEMM), 8k (GEMV), 4096, 0.64% (SpMM). . . . .	54
3.19	<i>Left</i> : A synthetic parallel application that launches threads to process $N \times N$ matrices. Each thread ( $i$ ) reads the input value and bins it into one of $D$ bins, ( $ii$ ) applies $R$ instances of function $f_d$ unique to bin $d$ and writes the result. Each element of a <i>coefficient</i> array feeds into $f_d$ . Thus the input is reused $R$ times and the degree of divergence scales with $D$ . <i>Right</i> : Speedup of Transmuter with a uniform-random matrix (# GPEs = # GPU threads = 64). Transmuter reconfigures from Trans-PS to Trans-SC beyond $R = 4$ . . . . .	55
4.1	Overview of the proposed CoSPARSE framework. . . . .	61

4.2	Structure of CoSPARSE hardware and software reconfiguration framework. For every invocation to CoSPARSE, we select the best software (inner product or outer product), followed by hardware configurations (Trans-SCS or Trans-SC for inner product, Trans-PC or Trans-PS for outer product), assuming a $2 \times 4$ system. . . . .	64
4.3	Matrix partitioning based on NZEs and algorithm mapping of inner product on Trans-SCS and outer product on Trans-PS that focuses on maximizing data reuse and reducing stalls for random accesses on a $2 \times 2$ system. . . . .	65
4.4	Speedup of outer product (Trans-PC) vs. inner product (Trans-SC). Generally, inner product performs better for dense vectors and outer product performs better for sparse vectors. The crossover vector density decreases when more PEs are present in a tile. . . . .	68
4.5	Speedup of Trans-SC vs. Trans-SCS for inner product. Trans-SCS achieves more performance gain for denser vectors or when the reuse of data in SPM is higher. . . . .	69
4.6	Speedup of Trans-PC vs. Trans-PS for outer product. The performance gain of Trans-PS grows with increasing vector density, increasing number of tiles, and decreasing number of PEs per tile. . . . .	70
4.7	The SpMV execution time of power-law matrices normalized to uniform matrices on Trans-SC (inner product) and Trans-PC (outer product) on an $8 \times 16$ system. Workload balancing benefits inner product more than outer product, especially Trans-SC for inner product. . . . .	74
4.8	Speedup and energy efficiency gain of CoSPARSE ( $16 \times 16$ ) over CPU and GPU. The vector density sweeps from 0.001 to 1.0. CoSPARSE achieves an average speedup (energy efficiency gain) of $4.5 \times (282.5 \times)$ and $17.3 \times (730.6 \times)$ over CPU and GPU, respectively. . . . .	76
4.9	Vector density, execution time normalized to inner product in Trans-SC, and the best software/hardware configuration for each iteration of CoSPARSE ( $16 \times 16$ ) for SSSP on soc-pokec. Each iteration is color coded with the best configuration. The best configuration changes with the active vertex set, which conforms to the analysis in Sec. 4.3.3. . . . .	77
4.10	Speedup and efficiency gain of CoSPARSE ( $16 \times 16$ ) over Ligra (Intel Xeon E7-4860 at 2.6 GHz, 48 cores with 256GB DRAM). . . . .	78
5.1	Transposition and compressed storage formats for sparse matrices. . . . .	83
5.2	(a) Breakdown of SSSP execution time on CoSPARSE [54] for graph <i>amazon</i> based on common misconceptions, using <code>mergeTrans</code> [195], and using our work, MeNDA. (b) Execution time comparison of recent proposals for sparse matrix transposition ( <code>mergeTrans</code> ) and SpMM (OuterSPACE[151] / SpArch[214]). Recent hardware breakthroughs have greatly optimized sparse applications, <i>e.g.</i> SpMM and SpMV, whereas little research effort has been spent on accelerating sparse matrix transposition, making transposition an increasingly evident bottleneck. . . . .	85

5.3	(a) Roofline model of <code>mergeTrans</code> [195] running with 64 threads. Sparse matrix transposition is memory bandwidth bound because the data points are close to the "roof", <i>i.e.</i> the red and blue lines that label the peak throughputs which can be achieved when the system memory bandwidth is fully utilized. (b) Memory bandwidth utilized by <code>mergeTrans</code> with an increasing number of threads. The memory bandwidth utilization saturates before reaching maximum due to the bottleneck at the memory interface. . . . .	86
5.4	Dataflow of MeNDA performing transposition on the sparse matrix in Figure 5.1. Each round of merge sort is executed sequentially on a 4-way merge tree. Left: The outcome of each round in the dense data structure. Right: The real input and output data of each round that are stored in memory. The input and output data are stored in the compressed data storage formats (CSR/CSC), and the intermediate data are stored in COO. . . . .	87
5.5	Architecture of MeNDA (top) and a MeNDA PU (bottom). A PU consists of a merge tree, prefetch buffers, a controller, a request queue, and a memory interface unit. The extra units required to support SpMV, <i>i.e.</i> a delay buffer and floating point adders and multipliers, are highlighted in red boxes. . . . .	89
5.6	Timing diagram of data propagation for merge sort shown in Figure 5.1 on a 4-leaf merge tree assuming a memory latency of 3 cycles. The cycle number and the corresponding memory activities are shown in the bottom right table. End-of-line signal propagation is shown with red arrows. . . . .	91
5.7	Matrix partitioning across 4 ranks. . . . .	93
5.8	(a) Sample pseudo-code of CoSPARSE using the programming interface of MeNDA and (b) the microarchitecture of the hardware substrate of CoSPARSE with 2 processing tiles and 4 PEs per tile. . . . .	95
5.9	Experimental methodology for MeNDA. . . . .	97
5.10	Speedup of MeNDA over <code>scanTrans</code> and <code>mergeTrans</code> on CPU [195] and <code>cuSPARSE</code> on GPU. The red line labels the speedup of 1. . . . .	100
5.11	Execution time of SSSP on CoSPARSE for <code>amazon</code> without runtime transposition, with runtime transposition using <code>mergeTrans</code> , and with runtime transposition using MeNDA. CoSPARSE ( $\sim 2 \times$ Storage) avoids runtime transposition at the cost of storing two copies of the graph [54]. . . . .	101
5.12	The execution time of MeNDA applying different optimizations normalized to that of the baseline implementation. In the legend, "prefetch" refers to stall-reducing prefetching enabled, "coal" refers to request coalescing enabled, and the number refers to the size of the prefetch buffers. . . . .	102
5.13	Execution time and throughput of MeNDA sweeping matrix size and density and the number of channels. . . . .	103
5.14	The execution time of the uniform matrices compared with that of the power-law matrices with the same sizes and densities. . . . .	104
5.15	The execution time and EDP of MeNDA sweeping the accelerator frequency (left) and number of leaf PEs (right). . . . .	105
5.16	Energy efficiency gain of MeNDA over Sadi <i>et al.</i> [165] for SpMV. . . . .	106

## LIST OF TABLES

2.1	Specifications of the benchmarking desktop system. . . . .	7
2.2	Summary of application TLP and GPU utilization of all applications in the benchmarking suite. The color of the heat map region corresponding to $c_i$ indicates the percentage of time when $i$ threads are running concurrently. (*for PhoenixMiner, two packets were simultaneously executing on the GPU throughout the experiment) . . . . .	8
2.3	Transcode rate, TLP, and GPU utilization of WinX with and without NVIDIA CUDA/NVENC. Enabling the GPU improves the transcode rate and lowers the TLP. . . . .	11
3.1	Reconfigurable features at each level in Transmuter. In the “hybrid” memory mode, banks are split between caches and SPMs. . . . .	26
3.2	Critical host- and Transmuter-side C++ intrinsics used to write optimized kernel libraries (TID = Tile ID, GID = GPE ID). Note that the API is depicted for a single-cluster design, for simplicity. . . . .	35
3.3	Microarchitectural parameters of Transmuter gem5 model. . . . .	37
3.4	Specifications of baseline platforms and libraries evaluated. . . . .	38
3.5	Energy-efficiency improvements (black) and deteriorations (red) of Transmuter over prior FPGAs, CGRAs, and ASICs. . . . .	50
3.6	Power and area of a $64 \times 64$ Transmuter cluster in 14 nm. . . . .	51
3.7	Estimated speedups for the end-to-end workloads in Fig. 3.3. . . . .	51
3.8	Qualitative comparison with prior work. . . . .	57
4.1	Definitions of Matrix_Op and Vector_Op of Algorithms mapped to CoSPARSE, where Sp represents the adjacency sparse matrix and V represents the frontier vector. <i>src</i> is the source vertex and <i>dst</i> is the destination vertex. . . . .	72
4.2	Specifications for real-world graphs. . . . .	74
5.1	Parameters of Ramulator and MeNDA. . . . .	97
5.2	Specifications of CPU and GPU baselines. . . . .	98
5.3	Specifications of Synthetic Uniform* (N#) and Power-law†(p#) Matrices. . . .	99
5.4	Specifications of SuiteSparse Matrices [40]. . . . .	99

## LIST OF PROGRAMS

3.1	GEMV pseudocode on Transmuter in Trans-SC. . . . .	40
3.2	FFT pseudocode on Transmuter in Trans-SA. . . . .	43
3.3	SpMV pseudocode on Transmuter in Trans-SA. . . . .	44

## LIST OF ABBREVIATIONS

- API** Application Programming Interface
- ASIC** Application-Specific Integrated Circuit
- ASIP** Application-Specific Instruction-set Processor
- BFS** Breadth-First Search
- BLAS** Basic Linear Algebra Subprograms
- CAM** Content-Addressable Memory
- CF** Collaborative Filtering
- CGRA** Coarse-Grain Reconfigurable Architecture
- COO** Coordinate Format
- COTS** Commercial Off-The-Shelf
- CPU** Central Processing Unit
- CS** Critical Section
- CSC** Compressed Sparse Column Format
- CSR** Compressed Sparse Row Format
- DFG** Data-Flow Graph
- DIMM** Dual In-line Memory Module
- DMA** Direct Memory Access
- DMSpM** Dense Matrix - Sparse Matrix Multiplication
- DNN** Deep Neural Network
- DRAM** Dynamic Random Access Memory

**DSL** Domain-Specific Languages

**EDP** Energy-Delay Product

**FCFS-FR** First Come First Serve - First Ready

**FFT** Fast Fourier Transform

**FIFO** First-In-First-Out buffer

**FLOPS** Floating-Point Operations

**FP** Floating-Point

**FPGA** Field Programmable Gate Array

**FSM** Finite-State Machine

**FU** Functional Unit

**GEMM** General (dense) Matrix-Matrix multiplication

**GEMV** General (dense) Matrix-Vector multiplication

**GTEPS** Giga Traversed Edges Per Second

**GPE** General-purpose Processing Element

**GPP** General-Purpose Processor

**GPU** Graphics Processing Unit

**HBM** High-Bandwidth Memory

**HLL** High-Level Language

**HMC** Hybrid Memory Cube

**HPC** High-Performance Computing

**ISA** Instruction Set Architecture

**MAC** Multiply-And-Accumulate

**M-GEMM** Masked General Matrix - Matrix multiplication

**MIMD** Multiple-Instruction, Multiple Data

**ML** Machine Learning



**MLP** Memory-Level Parallelism

**MSB** Most Significant Bit

**MSHR** Miss Status Holding Register

**NMP** Near-Memory Processing

**NZE** Non-zero Element

**NNZ** Number of Non-Zeros

**LCP** Local Control Processor

**LRG** Least-Recently Granted

**LS** Load/Store

**OT** Optimal Transport

**PE** Processing Element

**POSIX** Portable Operating System Interface

**PR** PageRank

**PU** Processing Unit

**RTL** Register-Transfer Level

**SIMD** Single Instruction, Multiple Data

**SIMT** Single Instruction, Multiple Threads

**SM** Streaming Machine

**SMT** Simultaneous Multi-Threading

**SpMM** Sparse Matrix-Matrix Multiplication

**SpMV** Sparse Matrix-Vector Multiplication

**SPM** Scratchpad Memory

**SPMD** Single-Program, Multiple Data

**SRAM** Static Random Access Memory

**SSN** Swizzle-Switch Network

**SSSP** Single-Source Shortest Path

**TLP** Thread-Level Parallelism

**VR** Virtual Reality

**XCU** crosspoint control unit

## ABSTRACT

Recent years have witnessed a tremendous surge of interest in accelerating sparse linear algebra applications. Sparse linear algebra is a fundamental building block and usually the performance bottleneck of a wide range of applications, such as machine learning, graph processing, and scientific computing. Optimizing sparse linear algebra kernels is thus critical for the efficient computation of these workloads. The key challenge of sparse linear algebra lies in the irregular access pattern induced by the sparseness nature, which renders the deep cache hierarchy in General-Purpose Processors (GPPs) useless and makes sparse linear algebra applications notoriously memory intensive. This dissertation proposes multiple approaches to optimize the performance and efficiency of sparse linear algebra kernels by taking advantage of emerging architecture techniques, including hardware specialization, architecture reconfiguration, and Near-Memory Processing (NMP).

Aiming for a balance among efficiency, flexibility, and programmability for architecture designs, this dissertation first proposes Transmuter, a reconfigurable architecture that features massively-parallel Processing Elements (PEs) and a flexible on-chip memory hierarchy that reconfigures the memory type, resource sharing, and dataflow at runtime to adapt to different applications. Transmuter demonstrates significant efficiency gains over the Central Processing Unit (CPU) and Graphics Processing Unit (GPU) across a diverse set of commonly-used kernels while offering GPU-like programmability. More importantly, Transmuter retains high performance for sparse linear algebra kernels, achieving an energy efficiency within  $4.1\times$  compared to state-of-the-art functionally-equivalent Application-Specific Integrated Circuits (ASICs).

As the algorithm mapping and hardware configuration play a crucial role in the performance of Transmuter, the next piece of this dissertation proposes the CoSPARSE framework, which guides the runtime reconfiguration of Transmuter to achieve the best performance for graph analytics workloads. During execution, CoSPARSE intelligently reconfigures to the best-performing software algorithm and hardware configuration for Transmuter based on the input characteristics. The synergistic software and hardware reconfiguration

amass a net speedup of up to  $2.0\times$ , over a naïve baseline implementation with no software or hardware reconfiguration. Compared to a recent graph processing framework on a server-class CPU, CoSPARSE achieves an average speedup and energy efficiency improvement of  $1.5\times$  and  $404.4\times$ , respectively, across a suite of widely-used graph algorithms.

The dynamic algorithm reconfiguration of CoSPARSE and many other graph frameworks requires the input graph to be stored in multiple data formats to avoid runtime transposition, trading off storage for performance. As data sizes keep growing, to prevent designs like CoSPARSE from expensive disk accesses when memory storage is limited, the final part of this dissertation presents MeNDA, a scalable near-memory accelerator for sparse matrix transposition and sparse merging dataflows. The wide application of multi-way merge sorting allows MeNDA to be easily adapted to other sparse primitives such as Sparse Matrix-Vector Multiplication (SpMV). Compared to two state-of-the-art implementations of sparse matrix transposition on a CPU and a sparse library on a GPU, MeNDA achieves a speedup of  $19.1\times$ ,  $12.0\times$ , and  $7.7\times$ , respectively. Because MeNDA greatly reduces the runtime transposition overhead, integrating MeNDA can save reconfigurable graph frameworks such as CoSPARSE from storing two or more copies of the graph in the main memory with a minor power overhead of 78.6 mW per rank for commodity Dynamic Random Access Memory (DRAM) devices.

# CHAPTER 1

## Introduction

The emergence of big data and massive social networks, as well as the prevalence of artificial intelligence, have resulted in increasing interest in sparse linear algebra applications. Today, sparse linear algebra is prevailing in a wide variety of important application domains, such as machine learning, graph analytics, scientific computing, *etc* [10, 80, 151, 155]. Sparse linear algebra refers to linear algebra routines on data structures, where a significant number of data elements are zero. To avoid redundant storage and computations on the zero elements, the sparse data structures are often stored in compressed data formats, which store only indices and values of the Non-zero Elements (NZE) [34]. The performance of sparse linear algebra computation thus relies heavily on efficient encoding/decoding of the compressed data formats when traversing the sparse data structure. The distribution of the NZEs in real-world sparse data structures tends to be irregular, which often results in uneven work distribution on hardware units, making parallelizing sparse linear algebra computation even harder [54]. Improving the performance and efficiency of sparse linear algebra applications is challenging, and therefore requires not only algorithms that minimize redundant computations and data accesses but also architecture designs that take full advantage of the algorithmic benefits with minimal overhead [151].

### 1.1 Emerging Architecture Techniques

The impending demise of Moore's law coupled with the end of Dennard scaling have led to the appearance of dark silicon (transistor under-utilization) [48] and consequently a surge in more specialized, application-specific architecture designs. As shown in Figure 1.1, the saturation of single-thread performance and the limitation in power consumption have made it difficult for chips to achieve the same performance benefits through advancements in process technology as before. Therefore, to extract more performance from a limited number of active on-chip transistors, computer architects gravitated towards heterogeneous

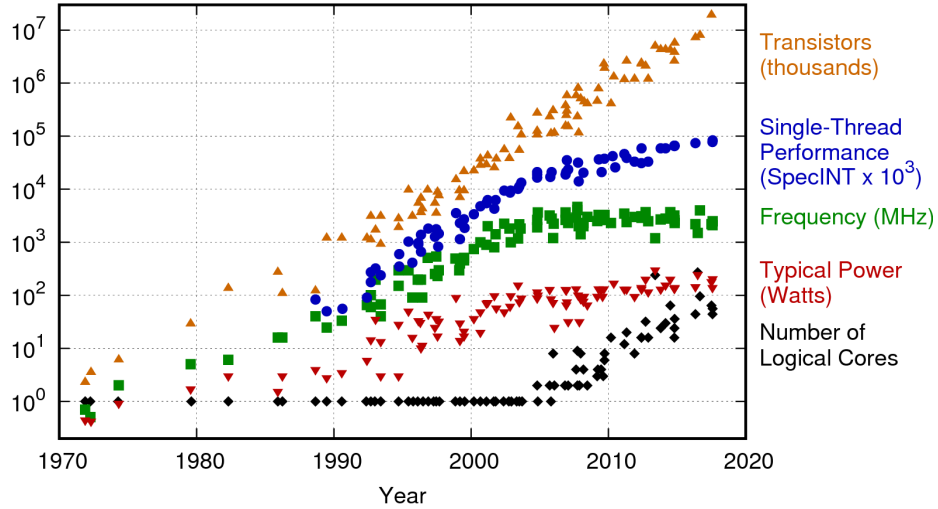


Figure 1.1: Growing trend in the number of transistors and microprocessor performance and power over the past 42 years [164].

computing systems that couple GPPs with fixed-function accelerators and are turning to ASIC for further improvements in performance and efficiency. Because of the irregular access pattern of sparse linear algebra applications induced by the sparseness nature, which renders the deep cache hierarchy in GPPs useless, the increasing interest in sparse linear algebra has led to a plethora of architecture proposals aiming to accelerate key computation kernels such as Sparse Matrix-Matrix Multiplication (SpMM) and SpMV [151, 214, 176, 10, 175, 79, 9, 165, 54, 80, 211].

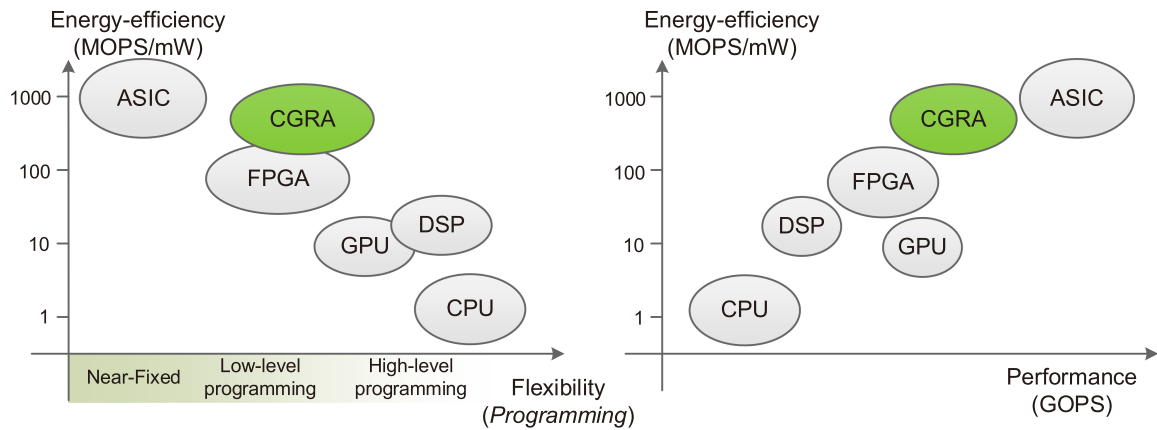


Figure 1.2: Comparison in performance, efficiency, and programmability/flexibility for different architecture designs [129].

Figure 1.2 demonstrates the performance, efficiency, and programmability of architecture designs with different levels of specialization. Despite the tremendous performance

and efficiency gains provided by ASICs, the fast pace of algorithmic innovation has made the short lifecycle and non-recurring engineering cost of ASICs a rising concern [129]. In addition, the stagnating on-chip transistor budget due to the end of Moore’s law will eventually limit the optimization space of ASICs, leading to diminishing hardware specialization returns [61]. Therefore, flexibility has also become an important consideration in architecture design, and many Coarse-Grain Reconfigurable Architectures (CGRAs) were proposed to strive for a balance between performance and efficiency as well as flexibility, in terms of software programmability or hardware reconfigurability [64, 162, 72, 147, 36].

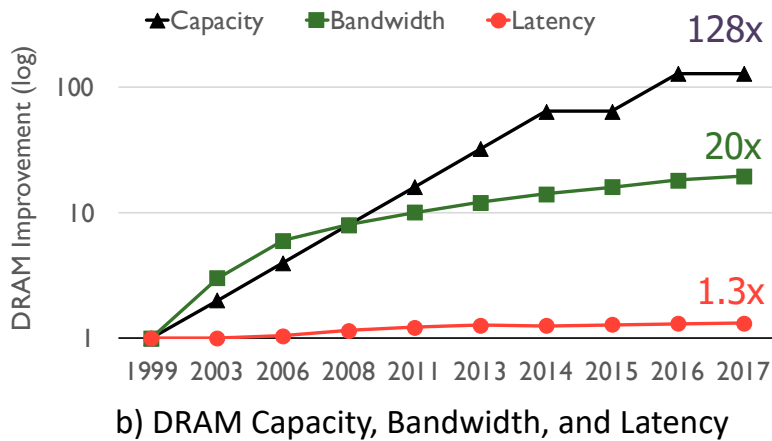
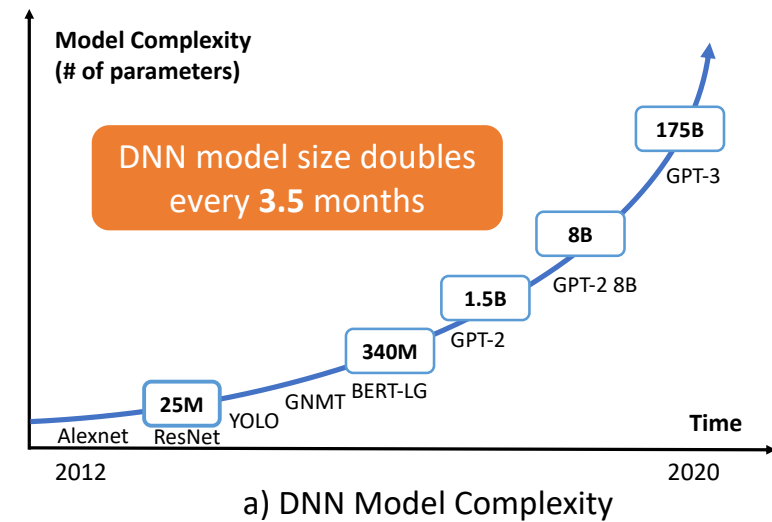


Figure 1.3: The growing trend of (a) DNN model complexity over the past 8 years [82] and (b) DRAM capacity, bandwidth, and latency over the past 20 years [143].

The arrival of big data has also posed great challenges to computer architecture studies. Figure 1.3(a) presents the growth of Deep Neural Network (DNN) model size over time. It is concluded that the DNN model size doubles every 3.5 months, which is much faster than the growth rate of memory size in modern computing systems. However, as shown

in Figure 1.3(b), the DRAM capacity doubles only every three years, while the trend is even worse for DRAM bandwidth and latency [143]. With the main memory becoming an increasingly significant bottleneck, exploring the available data locality and maximizing the on-chip data reuse has become the main focus of many ASIC designs, especially those targeting sparse linear algebra [151, 80, 155, 214, 176, 10, 175, 79, 9, 54, 80, 211]. The main memory bottleneck, which is majorly caused by the high energy and latency cost associated with data movement, has also given rise to memory-centric architecture designs, where computations are performed by processing logic inside the memory subsystem or the memory itself [143]. As sparse linear algebra is famous for its memory-bound nature resulting from the irregular memory access pattern, in-memory processing and NMP has been widely used for sparse linear algebra applications, such as sparse gathering in recommender systems [117, 97, 98, 8], and graph analytics [165, 200, 5, 144, 37, 213, 218].

## 1.2 Dissertation Overview

This dissertation proposes to accelerate sparse linear algebra workloads by taking advantage of these emerging architecture techniques, specifically hardware specialization, architecture reconfiguration, and NMP. To motivate the need for these emerging architecture techniques to accelerate sparse linear algebra, this dissertation starts with a parallelism analysis of contemporary hardware to understand the bottleneck and trend in hardware architecture designs. The analysis is then followed by a deeper discussion of the key challenges posed by sparse linear algebra applications on contemporary hardware.

To efficiently support the diverse and irregular sparse linear algebra kernels, this dissertation first proposes a reconfigurable, programmable architecture named Transmuter. Transmuter consists of massively parallel lightweight General-purpose Processing Elements (GPEs) that support a Commercial Off-The-Shelf (COTS) Instruction Set Architecture (ISA). A fabric of reconfigurable crossbars and reconfigurable caches connects the GPEs and the main memory and enables on-the-fly reconfiguration of the on-chip memory types, the resource sharing, and the dataflow. To ease the efforts of programming on Transmuter, a software stack is integrated to provide high-level python Application Programming Interfaces (APIs) for end users and low-level C++ APIs for library programmers. Evaluation shows that Transmuter achieves an efficiency gain of  $59.5\times$  and  $10.7\times$  over the CPU and GPU across a diverse set of kernels, respectively. While offering GPU-like programmability, Transmuter retains high performance on sparse linear algebra kernels, presenting an efficiency within  $4.1\times$  on average compared to functional equivalent ASICs.

Next, CoSPARSE, a software and hardware reconfigurable framework is developed to



further enhance the performance of Transmuter on SpMV and graph analytics workloads. CoSPARSE embeds SpMV implementations in the framework and automatically selects and switches to the best-performing software algorithm and hardware configuration upon an invocation of the SpMV routine. The reconfiguration decisions are heuristically driven based on extensive experiments on inputs with different sizes and densities. The combined software and hardware reconfiguration is able to achieve a speedup of up to  $2.0\times$  across the evaluated benchmarks compared to a naïve baseline with no reconfiguration. To implement a graph analytics algorithm, end users only need to define the key computations, similar to existing graph frameworks. For end-to-end graph processing workloads, CoSPARSE obtains a maximum speedup of  $3.51\times$  and a maximum energy efficiency gain of  $877\times$  compared to a state-of-the-art graph framework on a server-class CPU.

Finally, this dissertation presents MeNDA, a scalable near-DRAM multi-way merge solution for sparse matrix transposition and sparse merging dataflows. MeNDA deploys custom rank-level Processing Units (PUs) in the data buffer chips to expose the high-internal memory bandwidth of DRAM devices while limiting hardware modifications within the buffer chips. Because sparse data merging is widely applied in sparse linear algebra, MeNDA can be easily extended to support other sparse kernels, such as SpMV. Compared to two recently proposed sparse matrix transposition implementations on the CPU and a sparse library implementation on the GPU, MeNDA achieves an average speedup of  $19.1\times$ ,  $12.0\times$ , and  $7.7\times$ , respectively, while introducing a minor overhead of 78.6 mW per DRAM rank. More importantly, as MeNDA significantly reduces graph transposition overhead, graph frameworks like CoSPARSE can perform runtime graph transposition at algorithm reconfiguration with a minor latency overhead to avoid storing two or more copies of the input graph in different formats.

The rest of the document is organized as follows. Chapter 2 studies the parallelism of modern applications on contemporary architectures as well as the characteristics of sparse linear algebra workloads, motivating the need for emerging architecture techniques to accelerate sparse linear algebra. Chapter 3 presents the design details of Transmuter, a flexible architecture that efficiently supports a variety of kernels through runtime memory and dataflow reconfiguration, and evaluates it on a set of diverse kernels. Chapter 4 describes CoSPARSE, which enhances Transmuter by integrating an intelligent framework by judiciously choosing the best-performing algorithm and hardware configurations during execution for SpMV and graph analytics. Chapter 5 presents MeNDA a near-memory solution for sparse matrix transposition and sparse data merging, aiming to further improve the performance of sparse linear algebra applications by taking advantage of the high internal memory bandwidth of commodity DRAM devices. Chapter 6 concludes the document.

## CHAPTER 2

# Challenges of Executing Sparse Linear Algebra on Contemporary Hardware

To justify the need for emerging architecture techniques to accelerate sparse linear algebra applications, this chapter first presents a parallelism analysis on a high-end desktop computer to showcase the limitation of contemporary general-purpose computing systems and the hardware architecture design trend for emerging desktop applications. Then, a further discussion on the characteristics of sparse linear algebra is presented to explain the challenges of executing sparse linear algebra applications efficiently on GPPs as well as the opportunities provided by emerging computer architectures.

### 2.1 A Parallelism Analysis on Prominent Desktop Applications

Innovation in the domain of improving single-threaded performance hit a plateau in the early 21<sup>st</sup> century. Anticipating the end of Dennard scaling, which states that the power density of a chip remains almost constant across technology nodes [42], the hardware industry swiftly pivoted towards multi-core processors. The post-Dennard scaling era is plagued by the problem of dark silicon because improvements in cooling technology failed to keep up with the thermal design power requirements of newer technology node chips [48]. Today, the hardware world is experiencing a move to specialization, trading away silicon area for gains in energy efficiency [184, 151]. Modern systems are almost ubiquitously heterogeneous, involving a combination of the CPU with a GPU and/or fixed-function accelerators. Common desktop systems at homes and offices have 4-8 logical CPUs with a discrete GPU connected via PCI-Express.

To analyze how software has evolved to reap the benefits of multi-core and heterogeneous computers, we provide an 18-year perspective on the evolution of parallelism in desk-

top workloads based on studies on state-of-the-art systems in 2000 [56, 57] and 2010 [22]. We study a wide spectrum of commonly-used applications on a high-end desktop machine and analyze two important metrics, Thread-Level Parallelism (TLP) and GPU utilization. The work presented in this section was published in the form of a paper at ISPASS’19 [53].

### 2.1.1 Methodology

Table 2.1 shows the specifications of the evaluated system. TLP is defined in Equation 2.1, where  $c_i$  denotes the percentage of execution time when  $i$  threads are running simultaneously, in a system with  $n$  logical CPUs.  $c_0$  represents the idle time in the application.

$$\text{TLP} = \frac{\sum_{i=1}^n c_i i}{1 - c_0} \quad (2.1)$$

For GPU utilization, we consider the amount of time spent by work packets actually running over a period of time, where a packet is defined as a large collection of API calls packaged into a command stream. GPU utilization is measured by aggregating for all packets the ratio of packet running time to total time.

<b>CPU</b>	Intel Core i7-8700K, 3.70-4.70 GHz, 6 cores / 12 threads
<b>Graphics</b>	NVIDIA GTX 1080 Ti, 1481 MHz, 3584 CUDA cores
<b>RAM</b>	64 GB (16 GB × 4) DDR4 @ 3200 MHz
<b>Storage</b>	2 TB (1 TB × 2) PCIe NVMe SSD
<b>OS</b>	Windows 10 Education Version 1803

Table 2.1: Specifications of the benchmarking desktop system.

### 2.1.2 Evaluation Overview

Table 2.2 summarizes the TLP of the 6-core processor with Simultaneous Multi-Threading (SMT) enabled and the GPU utilization of the GTX 1080 Ti for all the applications. The “execution time” column illustrates the amount of time when 0, 1, ..., 12 logical cores are active simultaneously. The color of the heat map region corresponding to  $c_i$  illustrates the percentage of execution time when  $i$  threads are executed simultaneously. The “TLP” column shows the average of the TLP derived from 3 test iterations (with the same duration) for each application. Similarly, the “GPU utilization” column contains the average of the measured GPU utilization. Based on the low standard deviations from the 3 test iterations, we conclude that our experimental results are consistent. The last two columns in Table 2.2 present the average TLP and GPU utilization for each category, respectively.

Category	Application	Execution Time (%)												TLP	GPU Util (%)	Avg. TLP	Avg. GPU Util (%)
		C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11				
Image Authoring	Adobe Photoshop CC													8.6	1.6	4.2	6.8
	Autodesk Maya 3D 2019													2.7	9.9		
	Autodesk AutoCAD LT													1.2	9.0		
Office	Adobe Acrobat Pro DC													1.3	0.0	1.4	1.7
	Microsoft Excel 2016													2.1	2.1		
	Microsoft PowerPoint 2016													1.2	4.0		
	Microsoft Word 2016													1.3	1.7		
	Microsoft Outlook 2016													1.3	2.5		
Multimedia Playback	QuickTime Player 7.7.9													1.1	16.4	1.4	16.0
	Windows Media Player 12.0													1.3	16.1		
	VLC Media Player 3.0.3													1.8	15.7		
Video Authoring	CyberLink PowerDirector v16													4.3	6.3	3.1	3.4
	Adobe Premier Pro CC													1.8	0.6		
Video Transcoding	HandBrake 1.1.0													9.4	0.4	9.3	7.0
	WinX HD Video Converter 5.12.1													9.2	13.6		
Web Browsing	Firefox v60													2.2	8.6	2.1	5.9
	Chrome v60													2.2	5.1		
	Edge 42.17134.1.0													2.0	4.0		
VR Gaming	Arizona Sunshine 1.5.11046													3.4	68.2	3.1	76.3
	Fallout 4 VR 1.2													4.0	84.9		
	RAW Data 1.1.0													2.6	90.9		
	Serious Sam VR BFE 341433													2.4	72.2		
	Space Pirate Trainer 1.01													2.7	61.6		
	Project CARS 2 1.7.1.0													3.8	80.2		
Cryptocurrency Mining	Bitcoin Miner 1.54.0													5.4	98.9	4.8	98.7
	EasyMiner v.0.87													11.9	96.1		
	PhoenixMiner 3.0c													1.0	*100.0		
	Windows Ethereum Miner 1.5.27													1.0	99.7		
Personal Assistant	Cortana													1.4	2.7	1.3	1.4
	Braina 1.43													1.1	0.0		

Execution Time %

Table 2.2: Summary of application TLP and GPU utilization of all applications in the benchmarking suite. The color of the heat map region corresponding to  $c_i$  indicates the percentage of time when  $i$  threads are running concurrently. (\*for PhoenixMiner, two packets were simultaneously executing on the GPU throughout the experiment)

In summary, every application exploits parallelism to some extent, with a few applications showing more concurrency than others. For categories like office, multimedia playback, personal assistant, and web browsing, the degree of parallelism exploited is quite low, as concluded from the average TLP of around 2. Virtual Reality (VR) gaming displays moderate concurrency, with an average TLP ranging from 2 to 4. The TLP is expected to be similar within a category, but some categories are exceptions, including image authoring, video authoring, and cryptocurrency mining. There also exist applications that effectively utilize *most* of the available cores, e.g. applications for video transcoding exhibit an average TLP over 9. *Overall, the average TLP across all benchmarks is 3.1, where 6 out of 30 applications have an average TLP higher than 4.*

The values of GPU utilization are lower than 10% for most applications. Video authoring and transcoding applications exhibit moderate GPU usage. VR games and cryptocurrency miners, however, show significant utilization of the GPU, achieving an average GPU utilization of over 90%. *In general, the GPU is underutilized under most circumstances,*

except for graphic-intensive and cryptocurrency mining applications.

### 2.1.3 Evolution of Concurrency

The experimental results are compared to those collected from similar applications in prior work in 2000 [56, 57], and 2010 [22]. Figures 2.1 and 2.2 show the comparison of TLP and GPU utilization respectively. Although the TLP of benchmarks in media playback and video authoring has decreased (by 0.5-1.0), possibly due to the enhancements in single-core performance, most applications present either comparable or higher TLP. The significantly larger number of inputs (from sensors) and the escalation in computational complexity of VR games result in a noticeable rise in TLP compared to 3D games. Applications that have shown a large amount of concurrency in previous work, *e.g.* HandBrake, see a further increase in TLP. Even applications with little growth in average TLP exhibit progress. For example, Excel only has an average TLP of 2, yet its instantaneous TLP reaches the maximum of 12 during execution, which was not the case 8 years ago.

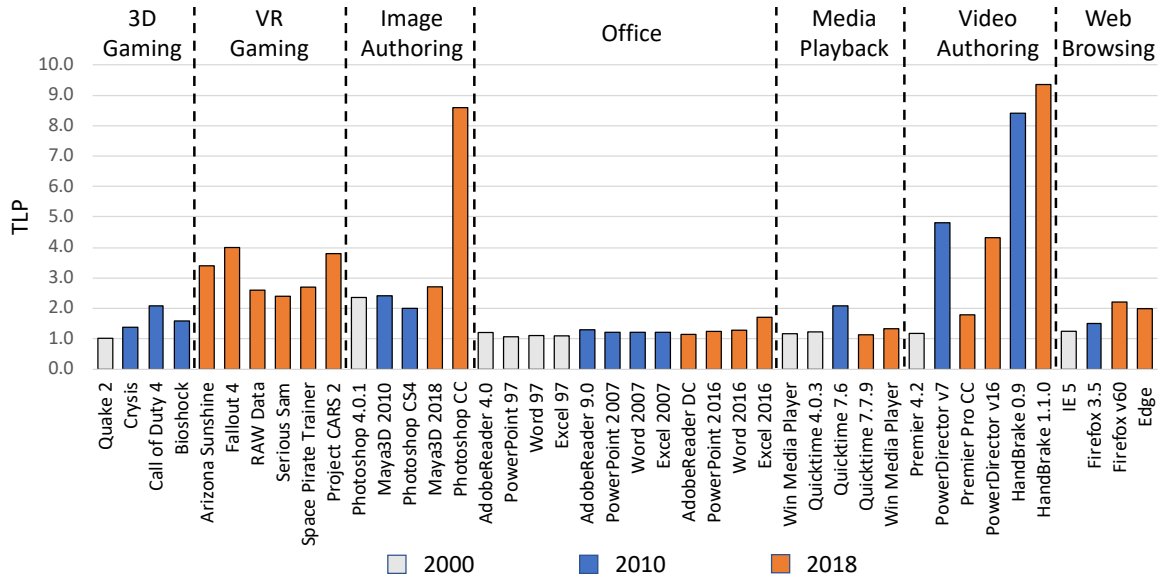


Figure 2.1: Comparison between TLP of desktop applications for 2000 [56, 57], 2010 [22], and 2018 [this work].

On the other hand, all benchmarks, except for those in VR gaming, show lower GPU utilization. This can be attributed to advancements in the GPU hardware, since a higher utilization of an older GPU, with fewer resources, is comparable to a lower utilization of a newer GPU, with more resources. The GPU utilization of VR games is commensurate with that of traditional 3D games. Since the current GPU has  $15\times$  more cores, a viable explanation is that the amount of offloaded work has also increased by an order of magnitude.

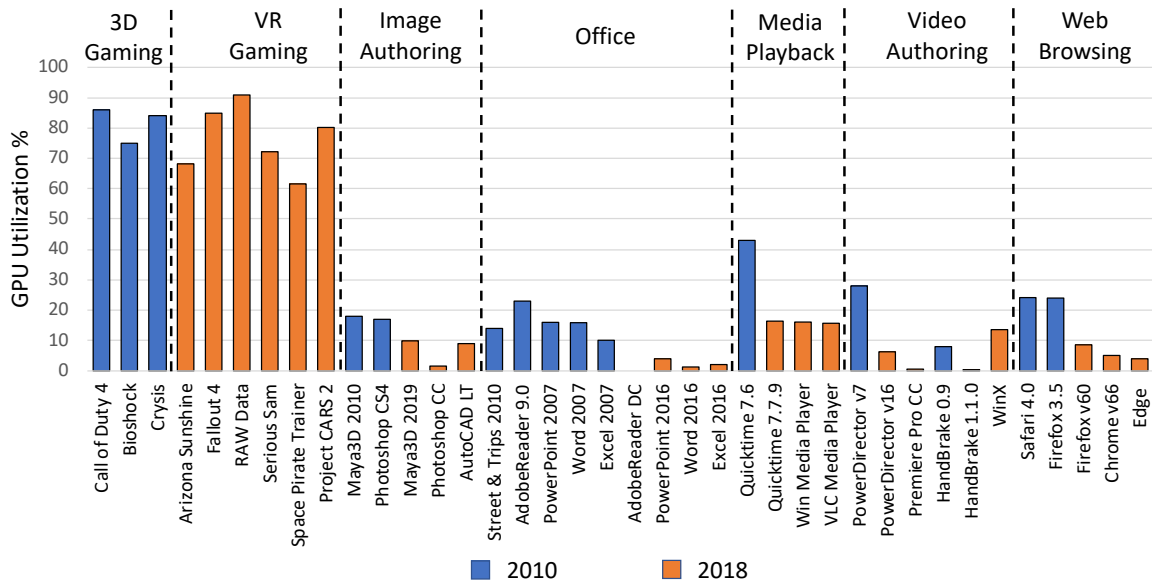


Figure 2.2: Comparison between GPU utilization of desktop applications for 2010 [22] and 2018 [this work].

## 2.1.4 Analyses on GPU

The dramatic breakthrough in GPU hardware over the past couple of decades has made it crucial to understand how GPUs are used to effectively assist compute-intensive tasks and whether GPUs are exploited to their full potential.

### 2.1.4.1 GPU Offloading

The performance and GPU utilization of HandBrake and WinX with the high-end GTX 1080 Ti and the mid-end GTX 680 are shown in Figure 2.3. HandBrake does not offload tasks to the GPU, so the GPU utilization stays below 1%, regardless of the number of active cores and the GPU settings. WinX, on the other hand, supports hardware acceleration with CUDA/NVENC. The transcode rates for different GPUs are almost the same (the plots for GTX 680 are omitted as they overlap with those for GTX 1080 Ti). In order to achieve similar performance, the GTX 680, which is inferior to the GTX 1080 Ti, harnesses a much higher GPU utilization. If we use an even lower-end GPU, we expect the GPU utilization to further increase, and the performance will start to degrade after the GPU utilization saturates at the maximum.

The transcode rate, TLP, and GPU utilization of WinX, with and without GPU acceleration, are shown in Table 2.3. During video transcoding, the CPU offloads compute-intensive transcoding tasks to the GPU through specific APIs, and the amount of offloading, indicated by the GPU utilization, grows almost linearly with the increase in TLP. With

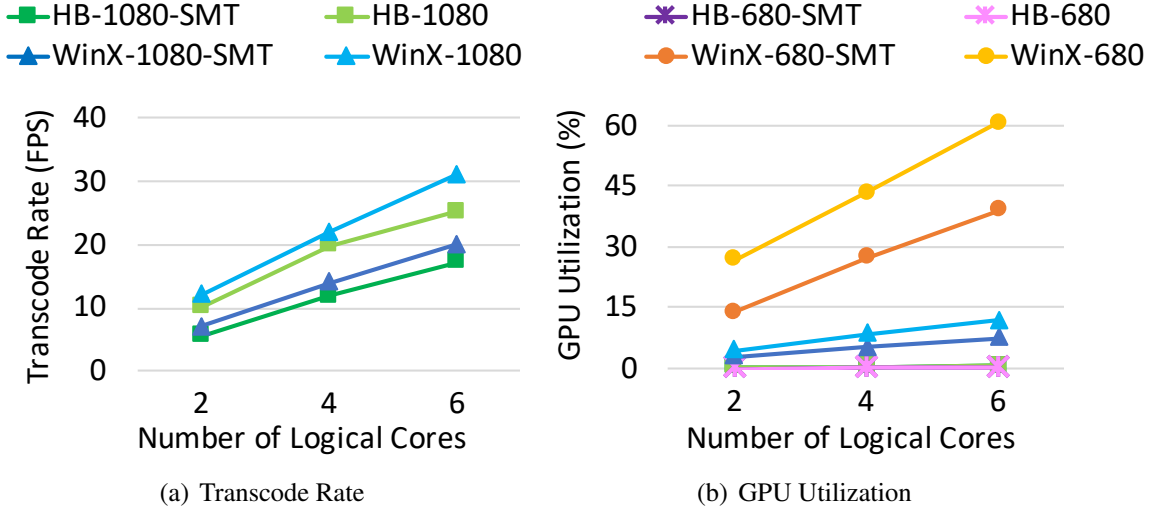


Figure 2.3: Transcode rates (GTX 1080 Ti only) and GPU utilization of HandBrake and WinX for 2, 4, and 6 logical cores, showing the effect of GPU offloading. The transcode rates for GTX 680 are not shown as they overlap exactly with those for GTX 1080 Ti.

Logical Cores	Transcode Rate		TLP		GPU Utilization (%)	
	No GPU	GPU	No GPU	GPU	No GPU	GPU
4	9	14	4.0	3.8	0.0	5.2
8	19	27	7.9	7.0	0.0	10.0
12	28	37	11.5	9.1	0.0	13.9

Table 2.3: Transcode rate, TLP, and GPU utilization of WinX with and without NVIDIA CUDA/NVENC. Enabling the GPU improves the transcode rate and lowers the TLP.

CUDA/NVENC enabled, the transcode rate of WinX improves by 143% on average and TLP decreases by up to 22%. *GPU acceleration not only increases performance but also relieves stress on the CPU, making it available for other tasks and protecting it from thermal throttling.* Similar offloading behavior is observed for Premiere Pro while exporting video with CUDA support, as shown in Figure 2.4. The assistance of GPU does not cause a significant change in runtime but slightly lowers the instantaneous TLP.

#### 2.1.4.2 GPU Utilization

As shown in Table 2.2, the GPU is under-utilized for most of the applications, which is possibly because the computational power of the GPU greatly exceeds what is demanded from it. The GPU indeed executes substantial tasks in various applications, such as hardware rendering in Maya and video export in PowerDirector, yet both exhibit GPU utilization lower than 10%. Even for WinX Video Converter, which uses CUDA/NVENC in the GPU

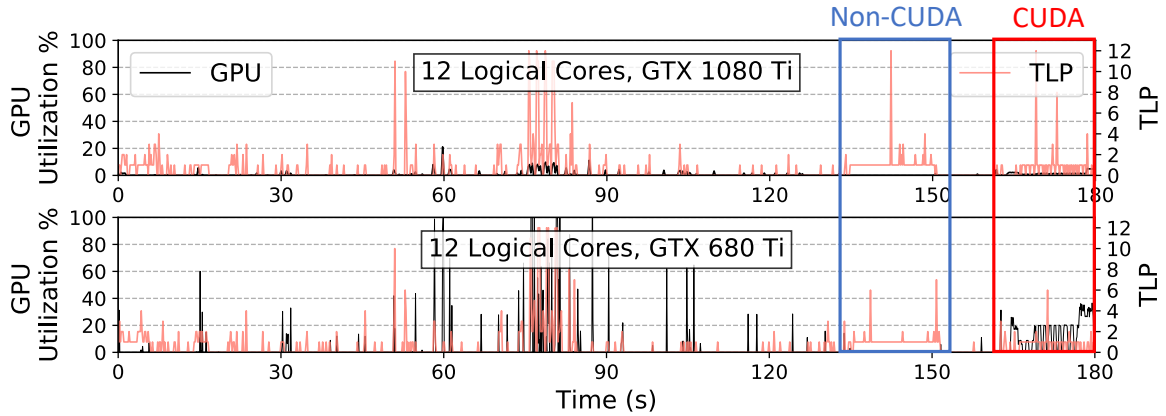


Figure 2.4: GPU utilization of GTX 680 and 1080 Ti for Premiere Pro. Video export with CUDA support shows higher utilization and lower TLP than without CUDA, and the utilization is higher for GTX 680.

for transcoding, the average GPU utilization is 13.6%. On the other hand, there are applications that utilize the GPU much more efficiently, such as VR games and cryptocurrency miners. We measure the GPU utilization of the mid-end GPU for video-related applications and cryptocurrency miners, as these use the GPU more than the others, and compare them to the utilization of the high-end GPU. The results are shown in Figure 2.4. Most applications see a notable improvement in utilization, except for cryptocurrency mining. Both GPUs show utilizations of up to 100% for Bitcoin Miner and EasyMiner, but as expected, the hash rate of GTX 680 is at least  $2\times$  lower despite the assistance of the CPU. Windows Ethereum Miner, however, has a higher GPU utilization with the superior GPU, since NVIDIA's Kepler architecture in GTX 680, released before the prevalence of cryptocurrency, is not optimized to run mining workloads.

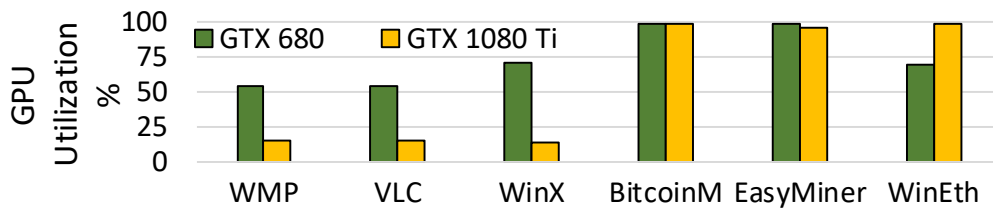


Figure 2.5: GPU utilization of GTX 680 and 1080 Ti for applications that show substantial use of GPU. VR is excluded as it requires a GPU better than GTX 970. PhoenixMiner does not support GTX 680.

In summary, a mid-end GPU is sufficient for most applications, including video editors and transcoders. However, for applications, such as VR gaming and mining, that perform intensive computations on the GPU, a high-end GPU is indispensable, as the mid-end GPU



causes a significant performance loss.

### 2.1.5 Takeaways

Major advancements have taken place in desktop hardware over the past decade. Our results showed that software has improved to take advantage of the parallelism available in the hardware compared to the work in 2010 [22]. Noticeable increases were seen in many applications, including those reputed for effective utilization of processor cores like Hand-Brake and Photoshop. For applications with slight changes in TLP over the past 18 years, efforts for exploiting available parallelism were exhibited by them achieving high instantaneous TLP during execution. For example, Excel spent 3.7% of the time using the maximum number of available logical cores concurrently, and web browsers have shifted from single-process models to multi-process models, resulting in better responsiveness and reliability. Emerging applications also demonstrated good utilization of hardware resources. The average TLP of VR gaming is twice that of traditional 3D gaming, and cryptocurrency miners involving CPU mining have a TLP higher than that of over 80% of the benchmarks. However, despite all the appreciable progress made by software in exploiting parallelism, the average TLP across all benchmarks is still 3.1, where 6 out of 30 applications have an average TLP higher than 4, while the maximum achievable TLP is 12.

On the other hand, the overall GPU utilization was lower than that observed in 2010. This showed that the improvements in the number of available hardware resources in the GPU have been growing at a faster pace than improvements in the parallelism harnessed by software. However, emerging workloads, e.g. VR games and cryptocurrency miners, exhibited great potential, as they fully exploited the computation power of the GPU.

The above observations imply that efforts in exploiting parallelism have hit a plateau on general-purpose platforms like CPUs but domain-specific hardware such as fixed-function units and GPUs shows promising performance acceleration, especially on emerging applications. This characterization hence further justifies the recent research efforts toward emerging hardware architectures such as ASIC, CGRA, and NMP.

## 2.2 Prevalence and Challenges of Sparse Linear Algebra

Sparse data structures are ubiquitous in modern applications that operate on big data. Recent years have witnessed a rapidly growing interest in applications involving sparse data structures, such as sparse neural networks [155] and graphs [181]. The key computation components of these applications are sparse linear algebra. One fundamental kernel in

sparse linear algebra is SpMM. SpMM is a significant building block of multiple algorithms prevalent in graph analytics, such as breadth-first search [67, 68], matching [159], graph contraction [25], peer pressure clustering [170], cycle detection [209], Markov clustering [188], and triangle counting [13]. It is also a key kernel in many scientific-computing applications. For example, SpMM is a performance bottleneck in the hybrid linear solver applying the Schur complement method [205] and algebraic multigrid methods [18]. Other computing applications, such as color intersection searching [94], context-free grammar parsing [159], finite element simulations based on domain decomposition [77], molecular dynamics [85], and interior point methods [96] also rely heavily on SpMM.

Among the applications characterized in the previous section, domains such as personal assistants and web browsing are likely to widely deploy sparse linear algebra. However, they tend to display low parallelism in not only TLP but also in GPU utilization. This suggests that it is challenging to exploit parallelism for sparse linear algebra on CPUs and GPUs simply from the software aspect due to the limitations in the hardware and emerging architectures are required to harvest further performance and efficiency improvements.

Sparse linear algebra tends to incur plenty of irregular data accesses due to the sparseness and the datasets usually have much greater sizes than the on-chip memories. As a widely-used representative of sparse linear algebra, SpMM exposes the major challenges of sparse linear algebra, which lie in efficiently indexing and traversing data stored in the sparse data structure and maximizing the on-chip data reuse. Real-world matrices are often large and extremely sparse, *e.g.* the adjacency sparse matrix representing Facebook friendships is of size  $1.08 \text{ B} \times 1.08 \text{ B}$  but with only 0.0003% NZEs [151]. Because the majority of elements are zeros, sparse matrices are typically stored in compressed formats to avoid redundant storage and computation on zero elements. Figure 2.6 gives a high-level overview of SpMM using the inner product algorithm, which is widely applied to General (dense) Matrix-Matrix multiplication (GEMM). In the inner product method, a row of the input matrix  $A$  is multiplied by a column of the input matrix  $B$  to produce a single element in the result matrix  $C$ . GEMM achieves great throughput through this algorithm because of the predictable access pattern. For SpMM, however, a significant portion of the execution time is spent on matching the indices of the two operands to find the NZEs with the same row or column indices. Due to matrix sparsity, the majority of the index matching failed and thus a large amount of memory bandwidth is wasted on fetching data that do not contribute to the final result. Hence, recent architecture proposals tend to apply either the outer product algorithm [151, 214] or the row-wise algorithm [175, 211] for SpMM to avoid index matching. However, decoding the compressed data formats for computation still results in irregular data accesses and the limited on-chip storage further leads to repetitive fetching of

the same data, worsening the memory bottleneck. The memory-bound nature of SpMM is shared by many sparse linear algebra kernels. Therefore, optimizing sparse linear algebra execution requires not only algorithms and scheduling that minimize redundant computations and data fetching and maximize on-chip data reuse but also architecture designs that efficiently support the proposed dataflows.

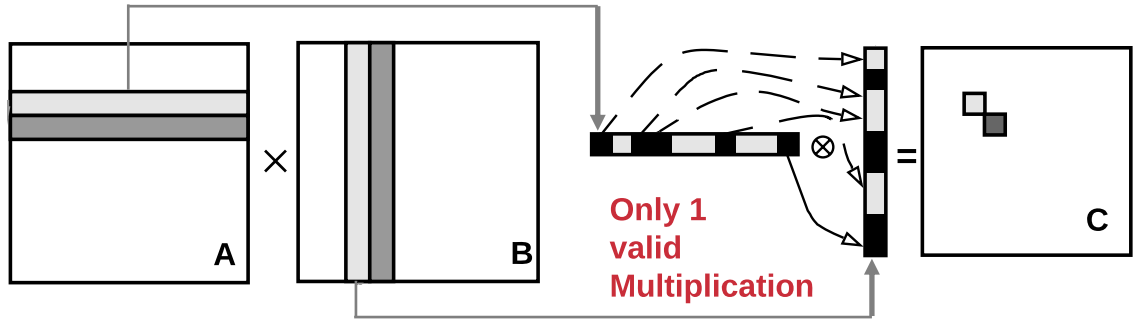


Figure 2.6: High-level overview of SpMM using the inner product algorithm [153]

Despite the great efforts spent on creating high-performance sparse linear algebra libraries for GPPs, *e.g.* Intel MKL and NVIDIA cuSPARSE, executing SpMM efficiently poses challenges on commodity general-purpose platforms, *i.e.* GPUs and CPUs [151]. Although CPUs can exploit memory-level parallelism and hide memory latencies through context switching among multiple threads, they are mainly optimized for complex instruction flows and therefore are equipped with complicated pipeline logic, such as advanced branch predictors, which introduces significant power and area overhead but does not help the performance of sparse linear algebra applications. The sparseness nature of the massive datasets also leads to a huge number of unpredictable memory accesses, rendering the traditional deep cache hierarchy designed to explore spatial and data locality useless. While GPUs demonstrate satisfactory compute efficiency on SpMV and dense matrix-matrix multiplication [19], compute units are significantly underutilized when the density drops below 0.1%, often achieving fewer than 1 GFLOPS [151], despite a peak theoretical throughput of over 4 TFLOPS [186]. This is further supported by the fact that rankings, such as the Green Graph 500 list [1], are dominated by CPU-based systems.

For large dense matrices, block partitioning and tiling techniques are used to take advantage of data locality [151]. However, when the density of the input matrices is decreased, the run-time is dominated by irregular memory accesses and index-matching in order to perform the multiplication. Moreover, while tiling techniques can reduce redundant reads to main memory in the short term, the on-chip storage constraints still necessitate that many data elements be redundantly fetched multiple times across tiles [90]. Finally, the irregu-

lar distribution of the NZEs further increases the difficulty of partitioning because of the potential workload imbalance issue [54].

As the efforts to extract more performance and efficiency from contemporary computing systems hit a plateau, many emerging computer architecture techniques have sprung up and shown great potential during the past decade. With the demise of Dennard scaling and Moore’s law, researchers have turned to ASICs for further performance and efficiency boosts [151]. Because accelerators can adopt custom architecture designs that best suit the desired dataflow, many accelerators have been proposed to satisfy the demands of various sparse linear algebra kernels [151, 214, 176, 10, 175, 79, 9, 165, 54, 80, 211]. On the other hand, accelerators remove redundant hardware logic at the cost of flexibility. As is similar to many other modern application domains, sparse linear algebra applications also feature diverse workload characteristics and fast-evolving algorithms. Hence, flexibility is no doubt a significant factor in architecture design for sparse linear algebra applications, in addition to performance and efficiency, which can be realized through software programmability or hardware reconfigurability [64, 162, 72, 147, 36]. Finally, with data movement becoming an increasingly significant bottleneck, NMP has gained a surge of interest because NMP reduces data movement and increases the effective system memory bandwidth by moving processing logic inside the memory subsystem [143]. Sparse linear algebra is known for its memory-bound nature and high data movement cost due to the large dataset size and the irregular memory access pattern, suggesting that sparse linear algebra has great potential to benefit substantially from NMP [117, 97, 98, 8, 165, 200, 5, 144, 37, 213, 218]. The stark inefficiencies on the hardware fronts and the opportunities suggested by the emerging architectures both motivate this work to formulate new approaches to accelerate sparse linear algebra kernels by exploring emerging architecture techniques.

## CHAPTER 3

# General-Purpose Acceleration through Reconfigurable Memory Hierarchy

As mentioned in Chapter 2, with the demise of Dennard’s scaling and Moore’s law, researchers have turned to ASICs for further performance and efficiency boost. Despite the superior performance and efficiency, ASICs compromise on generality and flexibility by removing extraneous hardware [152]. A naïve solution to cover the diverse algorithms and data characteristics in sparse linear algebra is to incorporate many ASICs in a system. However, if followed blindly, this approach will eventually hit the “accelerator wall” [61]. The fast-evolving algorithms and datasets also make accelerators subject to near-term obsolescence [152]. Therefore, the rising complexity in modern applications and the need for efficient and high-performance computing systems are urging for a solution that carefully trades off efficiency and flexibility.

To bridge the gap between efficiency and flexibility, this chapter presents Transmuter as a novel general-purpose architecture that adapts to the nature of the kernels through a flexible fabric of reconfigurable on-chip memory and interconnects. Transmuter supports compile-time and runtime reconfiguration of the on-chip memory type (cache/scratchpad), resource sharing (shared/private), and dataflow (demand-driven/systolic array) within 10s of nanoseconds. Each PE is designed to be an energy-efficient core that supports a standard ISA, allowing the architecture to be general-purpose and programmable. To further improve ease of adoption, a software stack is created so that drop-in replacements (*e.g.* Transpy) for standard libraries (*e.g.* Numpy) in host Python programs can be used to invoke embedded hand-optimized kernel implementations on Transmuter.

Transmuter is modeled using gem5 and the timing and power are validated against a simplified chip prototype. Iso-bandwidth/area comparisons demonstrate average throughput (energy-efficiency) improvements of  $5.0\times$  ( $18.4\times$ ) and  $4.2\times$  ( $4.0\times$ ) over a high-end CPU and GPU, respectively, across a diverse set of kernels used in graph analytics, scientific computing, machine learning, *etc.* More importantly, Transmuter achieves an average

energy efficiency of within  $9.3\times$  compared to state-of-the-art ASICs, while retaining support for arbitrary kernels. The work presented in this chapter was published in the form of an architecture paper at PACT’20 [152].

### 3.1 Introduction

The past decade has seen a surge in emerging applications that are composed of multiple kernels<sup>1</sup> with varying data movement and reuse patterns, in domains such as Machine Learning (ML), and graph, image, and signal processing. A growing number of such applications operate on compressed and *irregular* data structures [130, 93, 44], or on a combination of regular and irregular data [207, 43, 35]. While conventional GPPs generally suffice for desktop computing [53], areas such as High-Performance Computing (HPC) clusters and datacenters that demand higher performance for such applications require more specialized hardware; such systems are typically comprised of CPUs paired with GPUs and other domain-specific ASIC based accelerators [134, 91, 78], or Field Programmable Gate Arrays (FPGAs) [163, 149, 197]. CGRAs have also been proposed as promising alternatives for achieving near-ASIC performance [146, 183]. These platforms have been historically bounded by three conflicting constraints: *programmability*, *algorithm-specificity*, and *performance/efficiency* [129], as is illustrated in Fig. 3.1. Owing to these trade-offs, there is currently no single architecture that is the most efficient across a diverse set of workloads [154].

Thus, the rising complexity of modern applications and the need for efficient computing necessitate a solution that incorporates:

- **Flexibility.** Ability to cater to multiple applications, as well as emerging applications with changing algorithms, that operate on both regular and irregular data structures.
- **Reconfigurability.** Enabling near-ASIC efficiencies by morphing the hardware to specific kernel characteristics, for applications that are composed of multiple cascaded kernels.
- **Programmability.** Facilitating better adoption of non-GPP hardware by providing high-level software abstractions that are familiar to end-users and domain experts, and that mask the details of the underlying reconfigurable hardware.

To this end, we propose Transmuter, a reconfigurable accelerator that adapts to the nature of the kernel through a flexible fabric of lightweight cores, and reconfigurable memory

---

<sup>1</sup>This work refers to *kernels* as the building blocks of larger applications.

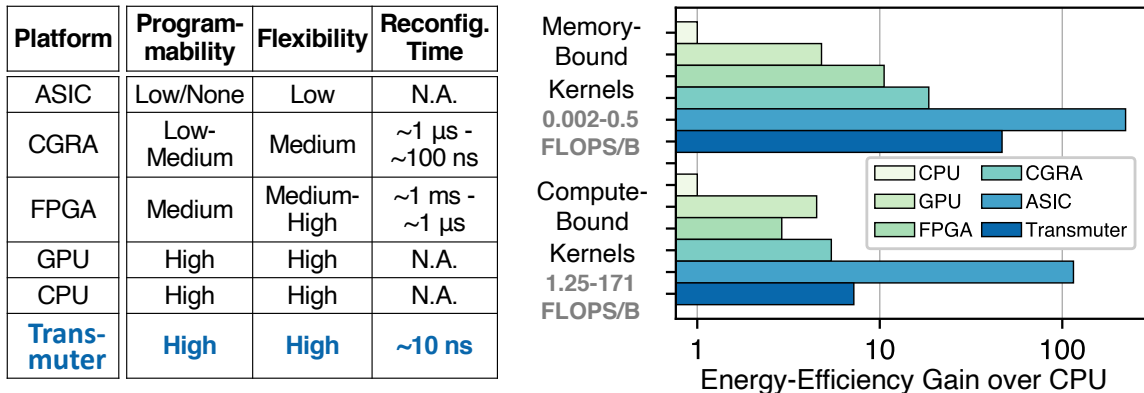


Figure 3.1: *Left*: Transmuter compared to contemporary platforms in terms of programmability, hardware flexibility, and reconfiguration overhead. *Right*: Energy-efficiency comparisons for kernels spanning a wide range of arithmetic intensities (FLOPS/B). Note that for ASICs and CGRAs, no single piece of hardware supports all kernels. Transmuter achieves  $2.0\times$  better average efficiency over state-of-the-art CGRAs while retaining the programmability of GPPs.

and interconnect. Worker cores are grouped into tiles that are each orchestrated by a control core. All cores support a standard ISA, thus allowing the hardware to be fully *kernel-agnostic*. Transmuter overcomes inefficiencies in vector processors such as GPUs for irregular applications [151] by employing a Multiple-Instruction, Multiple Data (MIMD) / Single-Program, Multiple Data (SPMD) paradigm. On-chip buffers and Scratchpad Memory (SPM) are used for low-cost scheduling, synchronization, and fast core-to-core data transfers. The cores interface to a High-Bandwidth Memory (HBM) through a two-level hierarchy of reconfigurable caches and crossbars.

Our approach fundamentally differs from existing solutions that employ gate-level reconfigurability (FPGAs) and core/pipeline-level reconfigurability (most CGRAs) — we reconfigure the on-chip *memory type*, *resource sharing*, and *dataflow*, at a coarser granularity than contemporary CGRAs, while employing general-purpose cores as the compute units. Moreover, Transmuter’s reconfigurable hardware enables run-time reconfiguration within 10s of nanoseconds, faster than existing CGRA and FPGA solutions (Sec. 3.2). Transmuter performs particularly well for sparse-data applications while remaining on average at least as efficient as a GPU for dense workloads. For mixed-data workloads, Transmuter rapidly reconfigures at run-time to cater to the nature of the application.

We further integrate a prototype software stack to abstract the reconfigurable Transmuter hardware and support ease of adoption. The stack exposes two layers: (i) a C++ intrinsics layer that compiles directly for the hardware using a COTS compiler, and (ii) a drop-in replacement for existing High-Level Language (HLL) libraries in Python, called

TransPy, that exposes optimized Transmuter kernel implementations to an end-user. Libraries akin to heterogeneous platforms (*e.g.* GPUs) are written by experts using the C++ intrinsics to access reconfigurable hardware elements. These libraries are then packaged and linked to existing HLL libraries, *e.g.* NumPy, SciPy, *etc.*

In summary, this work makes the following contributions:

- **Proposes a general-purpose, reconfigurable accelerator design** composed of a sea of parallel cores interweaved with a flexible cache-crossbar hierarchy that supports fast run-time reconfiguration of the memory type, resource sharing, and dataflow.
- **Demonstrates the flexibility of Transmuter** by mapping and analyzing six fundamental compute- and memory-bound kernels, that appear in multiple HPC and data-center applications, onto three distinct Transmuter configurations. A key takeaway is that the best configuration is dependent on the kernel as well as the input parameters.
- **Illustrates the significance of fast reconfiguration** by evaluating Transmuter on ten end-to-end applications (one in detail) spanning the domains of ML and graph / signal / image processing, that involve reconfiguration at kernel boundaries. Detailed analysis is presented for a representative mixed-data workload.
- **Proposes a prototyped compiler runtime and an HLL library called TransPy** that expose the Transmuter hardware to end-users through drop-in replacements for existing HLL libraries. The stack also comprises C++ intrinsics, which foster expert programmers to efficiently co-design new algorithms.
- **Evaluates the Transmuter hardware against existing platforms** with two proposed variants, namely TransX1 and TransX8, that are each comparable in area to a high-end CPU and GPU.

In summary, Transmuter demonstrates average energy-efficiency gains of  $18.4\times$ ,  $4.0\times$ ,  $3.4\times$ , and  $2.0\times$ , over a CPU, GPU, FPGAs, and CGRAs respectively, and remains within  $3.0\times$ - $32.1\times$  of state-of-the-art ASICs, while providing GPP-level programmability. Fig. 3.1 (right) presents a summary of these comparisons.

### 3.1.1 Contribution

The Transmuter architecture was developed and implemented in close collaboration with Subhankar Pal and Dong-hyeon Park for the Defense Advanced Research Projects Agency (DARPA) Software-Defined Hardware (SDH) program, with Subhankar Pal leading the architecture design and modeling. My main contributions lie in algorithm mapping, power



modeling, and performance evaluations. I implemented the kernel mappings in collaboration with Subhankar Pal (General (dense) Matrix-Vector multiplication (GEMV), GEMM), Dong-hyeon Park (SpMV, SpMM), Sung Kim (Fast Fourier Transform (FFT)) and Xin He (SpMV, SpMM in Trans-SA). The prototype software stack in Sec. 3.6 was developed by collaborators from the University of Edinburgh. The work presented in this chapter is mainly from our publication in PACT’20 [152].

## 3.2 Motivation

In this section, we first present some background on conventional approaches and their shortcomings in terms of bridging the gap between flexibility and performance. Next, we discuss the characterization studies of a suite of real-world applications that are composed of disparate kernels. We then analyze how inherent characteristics within these kernels motivate the choice of hardware and the need for hardware reconfigurability.

### 3.2.1 Contemporary Computing Platforms

Figure 3.2 shows a summary of prior studies that show trade-offs between performance/energy efficiency and flexibility/programmability across these architectural paradigms. CGRAs, FPGAs, and Application-Specific Instruction-set Processors (ASIPs) lie in the middle of the spectrum with GPPs and ASICs appearing at opposite ends.

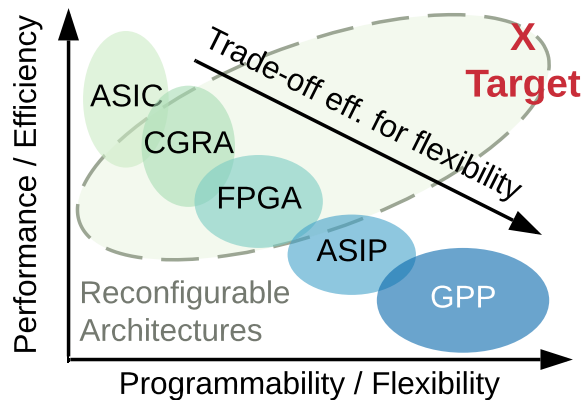


Figure 3.2: Trade-offs between programmability and efficiency in prominent computer architectures.

ASICs have been the subject of extensive research in the dark silicon era due to their superior performance and efficiency [169]. However, ASICs compromise on generality by stripping away extraneous hardware, such as control logic, thus limiting their functionality

to specific algorithms. An obvious solution is to design systems with multiple ASICs, (*e.g.* one per kernel) but that leads to high under-utilization for applications with cascaded kernels. In reality, emerging real-world problems seldom consist of just one algorithm or kernel (Sec. 3.2.2). Moreover, fast-moving domains, such as ML, involve algorithms that evolve faster than the turnaround time to fabricate and test new ASICs, despite efforts on accelerating the design flow [39], thus subjecting them to near-term obsolescence [29, 83]. Finally, ASICs are generally non-programmable, barring a few that use sophisticated software frameworks [2].

FPGAs have been successful in fast prototyping and deployment by eliminating non-recurring costs through programmable blocks and routing fabric. Moreover, high-level synthesis tools have reduced the low-level programmability challenges associated with deploying efficient FPGA-based designs [113, 112, 14]. Despite that, power and cost overheads prohibit FPGAs from adapting to scenarios that demand the acceleration of a diverse set of kernels [162, 28, 161]. Besides, reconfiguration overheads of FPGAs are in the ms- $\mu$ s range, even for partial reconfiguration [191, 201, 202], thus impeding fast run-time reconfiguration across kernel boundaries.

CGRAs overcome some of the energy and performance inefficiencies of FPGAs by reconfiguring at a coarser granularity. However, CGRA reconfiguration usually happens at compile-time, and the few that support run-time reconfiguration only support reconfiguration in the compute datapath [121], with overheads ranging from a few  $\mu$ s to 100s of ns [60, 64, 128]. Furthermore, many CGRAs require customized software stacks but have inadequate tool support, since they typically involve Domain-Specific Languages (DSL) and custom ISAs [198].

Finally, while CPUs and GPUs carry significant energy and area overheads compared to lean ASIC designs, they are the *de facto* choice for programmers as they provide high flexibility and abstracted programming semantics [24]. Although GPUs are efficient across many regular HPC applications, *i.e.* those exhibiting low control divergence, improving their effectiveness on irregular workloads remains a topic of research today [148, 26].

### 3.2.2 Taming the Diversity across Kernels

Many real-world workloads consist of multiple kernels that exhibit differing data access patterns and computational (arithmetic) intensities. In Fig. 3.3, we show the percentage execution times of key kernels that compose a set of ten workloads in the domains of ML, graph analytics, and signal/image/video processing. These workloads are derived from an ongoing multi-university program to study software-defined hardware.

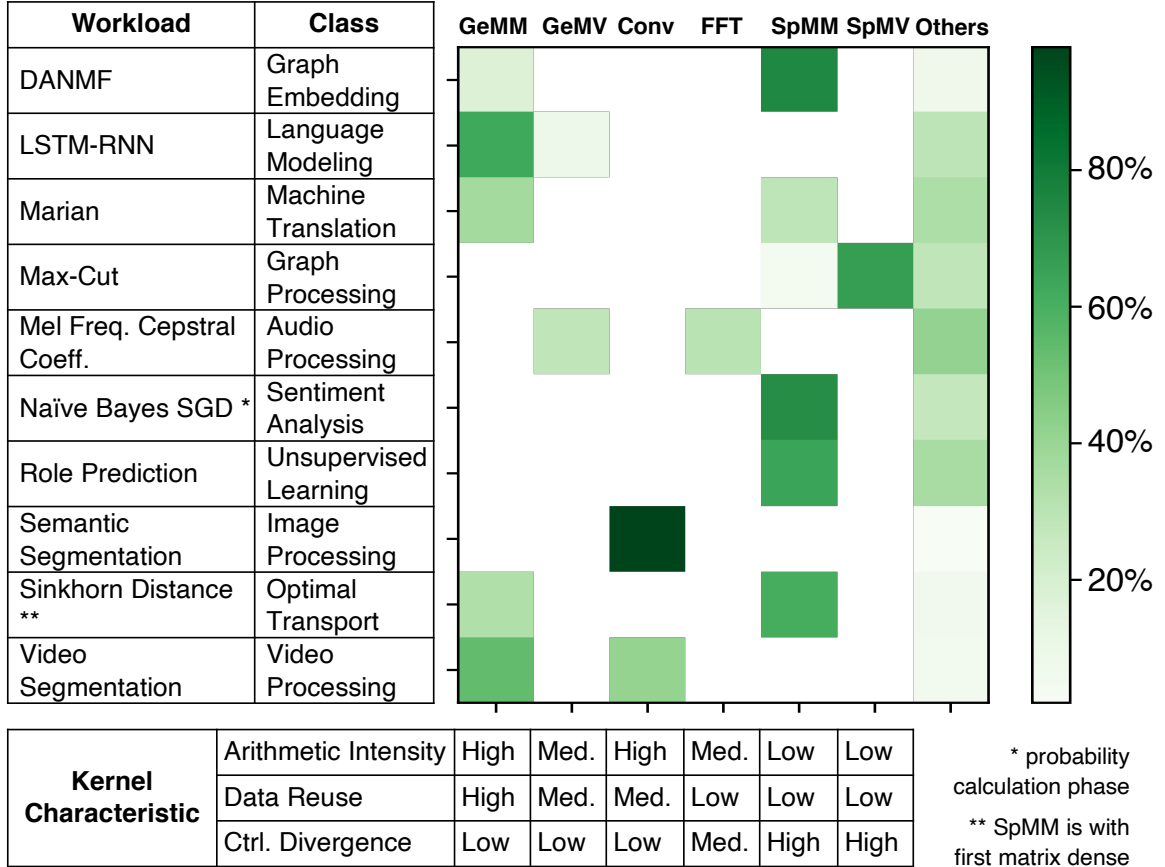


Figure 3.3: Fraction of execution time of kernels in applications spanning the domains of ML, signal processing, and graph analytics [207, 139, 93, 43, 130, 120, 44, 131, 35, 27] on a heterogeneous CPU-GPU platform. Some key characteristics, namely arithmetic intensity, data reuse, and divergence, of each kernel are also listed.

The underlying kernels exhibit a wide range of arithmetic intensities, from  $\frac{1}{1000}$ ths to 100s of floating-point operations per byte, *i.e.* Floating-Point Operations (FLOPS)/B (Fig. 3.1). We briefly introduce the kernels here. GEMM and GEMV are regular kernels in ML, data analytics, and graphics [51, 63]. Convolution is a critical component in image processing [4] and convolutional neural networks [109]. FFT is widely used in speech and image processing for signal transformation, with vast prior work on improving the speed and efficiency of FFT algorithms and architecture designs [140, 12]. SpMM is an important irregular kernel in graph analytics (part of GraphBLAS [100]), scientific computation [46, 18, 205], and problems involving big data with sparse connections [159, 85]. Another common sparse operation is SpMV, which is predominant in graph algorithms such as PageRank and Breadth-First Search [137], as well as ML-driven text analytics [7].

**Takeaways.** Fig. 3.3 illustrates that real-world applications exhibit diverse characteristics not only across domains but also within an application. One example of the latter

is *Sinkhorn*, an Optimal Transport (OT) application whose inner loop comprises GEMM followed by element-wise division with a sparse matrix, and dense matrix - sparse matrix multiplication. Thus, taming both the *inter*- and *intra*-application diversity efficiently in a *single piece of hardware* calls for an architecture capable of tailoring itself to the characteristics of each composing kernel.

### 3.2.3 Hardware Implication of Disparate Patterns

Intuition dictates that the diverse characteristics of kernels, as explored in Sec. 3.2.2, would demand an equivalent diversity in hardware. We study the implications of kernel characteristics on different hardware choices by analyzing the performance of synthetic microbenchmarks, and present our findings below.

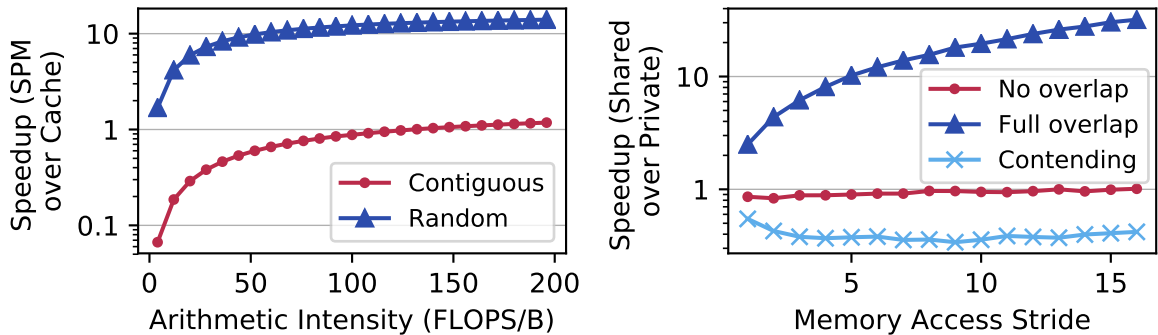


Figure 3.4: *Left*: Performance of an SPM over a cache-based single-core system for a synthetic workload with variable access patterns and arithmetic intensity. *Right*: Performance of a shared cache over a private cache-based 8-core system on a synthetic program with varying access strides and working set overlaps across cores. “Contending” is a case where all cores, in a given cycle, access addresses that map to the same bank in the shared mode.

**On-Chip Memory Type: Cache vs. Scratchpad.** Cache and SPM are two well-known and extensively researched types of on-chip memory [108, 16, 190]. To explore their trade-offs, we performed experiments on a single-core system that employs these memories. The results of the experiments are shown in Fig. 3.4 (left), and our observations are as follows:

- Workloads that exhibit low arithmetic intensity (*i.e.* are memory-intensive) but high spatial locality (*e.g.* contiguous memory accesses) favor a cache-based system.
- Workloads that are compute-intensive and have high traffic to disjoint memory locations favor an SPM *if* those addresses are known *a priori*. In this case, an SPM outperforms a cache because the software-managed SPM replacement policy supersedes any standard cache replacement policy.

Thus, caching is useful for kernels that exhibit high spatial locality and low-to-moderate FLOPS/byte, whereas SPMs are more efficient when the data is prone to thrashing, but is predictable and has sufficient reuse and is better managed by the programmer.

**On-Chip Resource Sharing: Private vs. Shared.** The performance of shared versus private on-chip resources is dependent on the working set sizes and overlaps across cores, *i.e.* inter-core data reuse. Specifically, we analyze the choice between and their accessibility to the compute fabric (**shared** and **private**). The amenability of these hardware features is heavily dependent upon two important kernel characteristics, namely arithmetic intensity (a measure of compute/memory-boundedness) and data locality. Fig. 3.4 (right) shows the trade-offs between sharing and privatizing the on-chip memories among cores in a multi-core, multi-banked system. From our experiments we noted:

- When there is significant overlap between the threads’ working sets, sharing leads to speedups exceeding  $10\times$  over privatization. This is owed to memory access coalescing and deduplication of data in the shared mode.
- When cores work on disjoint data, there is an insignificant difference in performance with sharing over no-sharing, if the union of the threads’ working sets fit on-chip.
- Regular kernels may exhibit strided accesses that can be hazardous for a shared multi-banked cache, due to conflicting accesses at the same bank. In this case, a private configuration delivers better performance. We illustrate this in the “contending” case, where all cores load/store with a stride that causes contention for a specific cache bank in the shared mode.

**Dataflow: Demand-Driven vs. Spatial.** In this work, we refer to demand-driven dataflow as the dataflow used by GPPs, wherein cores use on-demand loads/stores to read/write data and communicate via shared memory. In contrast, spatial dataflow architectures (*e.g.* systolic arrays) are data-parallel designs consisting of multiple PEs with direct PE-to-PE channels. Each PE receives data from its neighbor(s), performs an operation, and passes the result to its next neighbor(s) [110]. If pipelined correctly, this form of data orchestration harnesses the largest degree of parallelism. However, it is harder to map and write efficient software for certain applications on spatial architectures [89]. This is especially true for applications that operate on sparse data, where the inter-PE reuse is limited due to irregular memory accesses.

**Takeaways.** Through these observations, we derive the key insight that the on-chip *memory type*, *resource sharing*, and *dataflow* are three key hardware design choices that are each amenable to a different workload characteristic. This motivates the intuition that

an architecture that reconfigures between these designs can accelerate diverse workloads that exhibit a spectrum of characteristics. We thus propose Transmuter, a general-purpose accelerator that delivers high efficiency through dynamic reconfiguration (within 10 cycles) to tailor itself to the nature of the kernel.

### 3.3 Transmuter Overview

The takeaways from the previous section are the fundamental design principles behind our proposed architecture, Transmuter. In this section, we present an overview of our architecture and the reconfigurability features that enable it to cater to disparate workload characteristics.

Transmuter is a tiled architecture composed of a massively parallel fabric of simple cores. It has a two-level hierarchy of crossbars and on-chip memories that allows for fast reconfiguration of the on-chip *memory type* (cache/scratchpad/First-In-First-Out buffer (FIFO)), *resource sharing* (shared/private) and *dataflow* (demand-driven/spatial). The various modes of memory and dataflow configurations are listed in Table 3.1. The two levels of the memory hierarchy, *i.e.* L1 and L2, support 8 modes each. Furthermore, each Transmuter tile can be configured independently, however these tile-heterogeneous configurations are not evaluated in this work.

Table 3.1: Reconfigurable features at each level in Transmuter. In the “hybrid” memory mode, banks are split between caches and SPMs.

Dataflow	On-Chip Memory	Resource Sharing	# Modes
Demand-driven	Cache / SPM / Hybrid	Private / Shared	6
Spatial	FIFO + SPM	1D / 2D Systolic Sharing	2

In this work, we identify three distinct Transmuter configurations to be well-suited for the evaluated kernels based on characterization studies on existing platforms (Sec. 3.2.2). These configurations are shown in Fig. 3.5 and discussed here.

- **Shared Cache (Trans-SC).** Trans-SC uses shared caches in the L1 and L2. The crossbars connect the cores to the L1 memory banks and the tiles to the L2 banks, respectively, and perform arbitration when there is resource contention. This resembles a manycore system, but with a larger compute-to-cache ratio, and is efficient for workloads with regular data accesses and high inter-core reuse.
- **Private Scratchpad (Trans-PS).** Trans-PS reconfigures to private scratchpads in the L1, while retaining the shared caches in the L2. The crossbars privatize the L1 SPMs

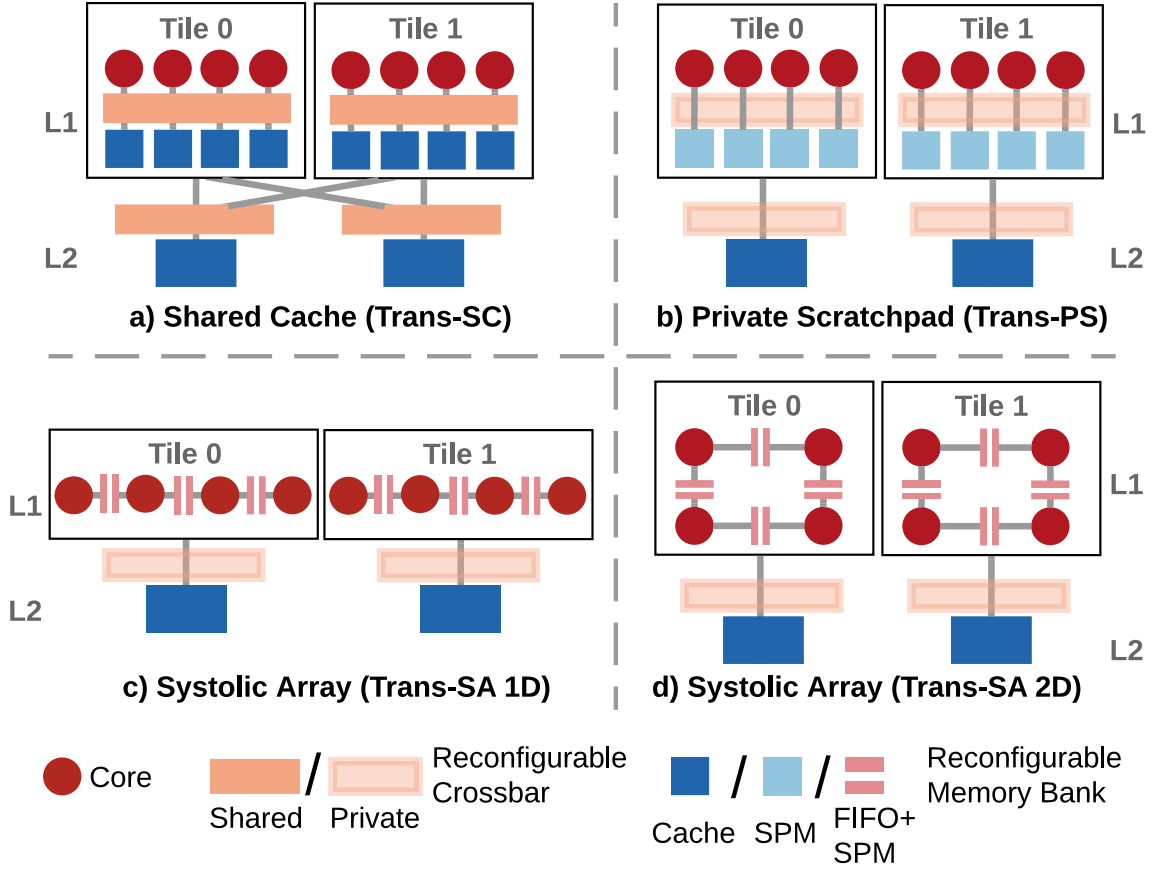


Figure 3.5: High-level Transmuter architecture showing the configurations evaluated in this work, namely a) Trans-SC (L1: shared cache, L2: shared cache), b) Trans-PS (L1: private SPM, L2: private cache), and c, d) Trans-SA (L1: systolic array, L2: private cache).

and the L2 cache banks to their corresponding cores and tiles, respectively. This configuration is suited for workloads with high intra-core but low inter-core reuse of data that is prone to cache-thrashing. The private L2 banks enable caching of secondary data, such as spill/fill variables.

- **Systolic Array (Trans-SA).** Trans-SA employs systolic connections between the cores within each tile and is suited for highly data-parallel applications where the work is relatively balanced between the cores. Transmuter supports both 1D and 2D systolic configurations. Note that the L2 is configured as a cache for the same reason as with Trans-PS.

While Transmuter supports a total of 64 configurations, we omit an exhaustive evaluation of all possible Transmuter configurations, given the space constraints of the paper. In the rest of the chapter, we use the notation of  $N_T \times N_G$  Transmuter to describe a system with  $N_T$  tiles and  $N_G$  GPEs per tile.

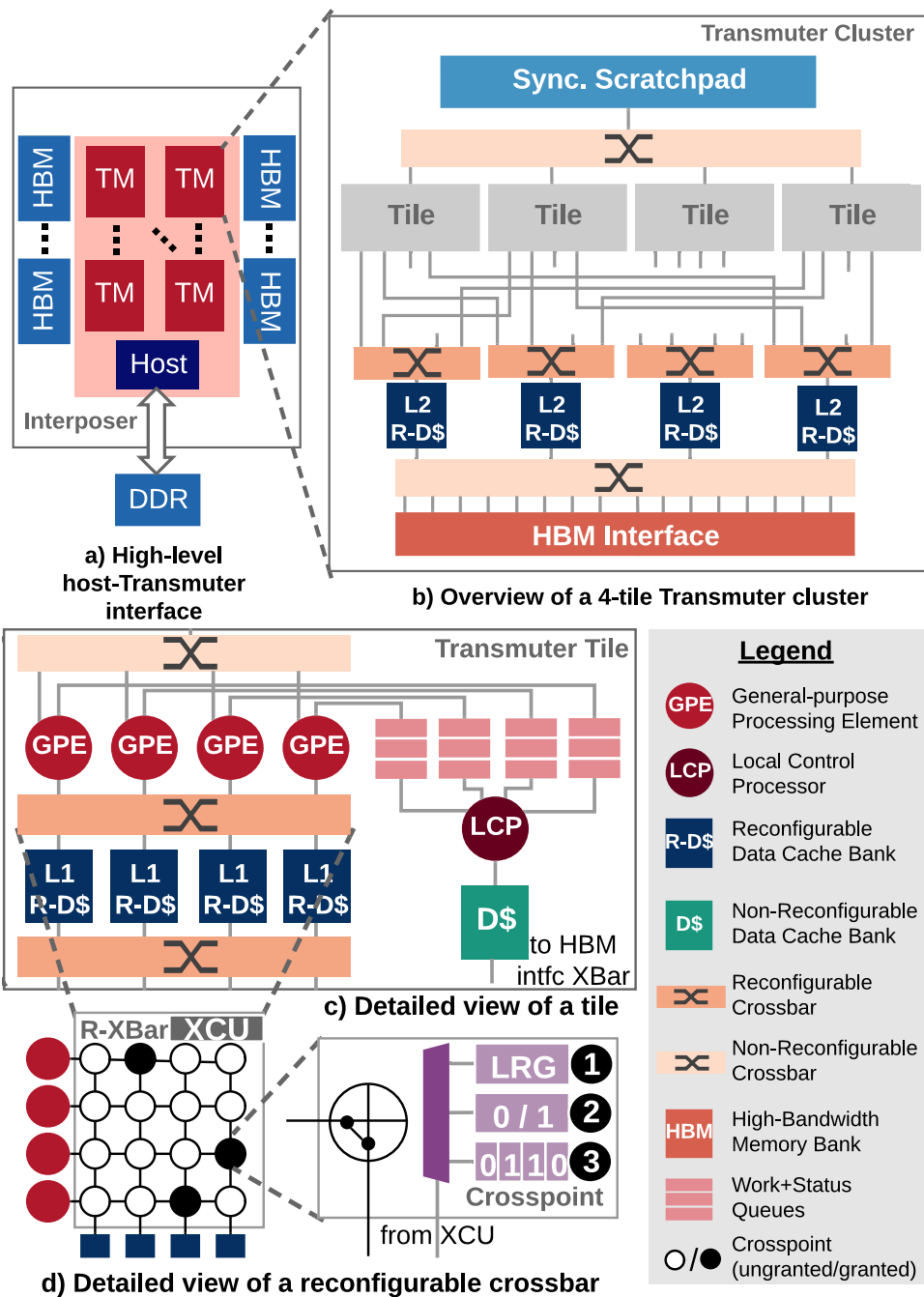


Figure 3.6: a) High-level overview of a host-Transmuter system. b) Transmuter architecture showing 4 tiles and 4 L2 R-DCache banks, along with L2 R-XBars, the synchronization SPM, and interface to off-chip memory. Some L2 R-XBAR input connections are omitted for clarity. c) View of a single tile, showing 4 GPEs and the work/status queues interface. Arbiters, instruction paths, and caches (ICaches) are not shown. d) Microarchitecture of an R-XBar, with the circled numbers indicating the mode of operation: ①: ARBITRATE, ②: TRANSPARENT, ③: ROTATE.



## 3.4 Transmuter Architecture Design

In this section, we detail the design of the Transmuter hardware, synchronization handling, and reconfiguration overheads. A full Transmuter system is shown in Fig. 3.6-a. A Transmuter chip consists of one or more Transmuter clusters interfaced to HBM stack(s) in a 2.5D configuration, similar to modern GPUs [119]. A small host processor sits within the chip to enable low-latency reconfiguration. It is interfaced with a separate DRAM module and data transfer is orchestrated through Direct Memory Access (DMA) controllers (not shown) [62]. The host is responsible for executing serial/latency-critical kernels, while parallelizable kernels are dispatched to Transmuter.

### 3.4.1 General-purpose Processing Element and Local Control Processor

A GPE is a small processor with Floating-Point (FP) and Load/Store (LS) units that uses a standard ISA. Its small footprint enables Transmuter to incorporate many such GPEs within standard reticle sizes. The large number of GPEs coupled with Miss Status Holding Registers (MSHRs) in the cache hierarchy allows Transmuter to exploit Memory-Level Parallelism (MLP) across the sea of cores. Minor modifications are made to the GPE pipelines to handle control hazards introduced due to a custom `PUSH/POP` interface (Sec. 3.4.2). The GPEs operate in a MIMD/SPMD fashion, and thus have private instruction (I-) caches.

GPEs are grouped into tiles and are coordinated by a small control processor, the Local Control Processor (LCP), which belongs to the same class of cores as the GPEs. Each LCP has private D- and ICaches that connect to the HBM interface. The LCP is primarily responsible for distributing work across GPEs, using either *static* (e.g. greedy) or *dynamic* scheduling (e.g. skipping GPEs with full queues), thus trading-off code complexity for workload balance.

### 3.4.2 Work and Status Queues

The LCP distributes work to the GPEs through private FIFO work queues. A GPE similarly publishes its status via private status queues that interface to the LCP (Fig. 3.6-c). The queues block when there are structural hazards, *i.e.* if a queue is empty and a consumer attempts a `POP`, the consumer is idled until a producer `PUSHes` to the queue, thus preventing wasted energy due to busy-waiting. This strategy is also used for systolic accesses, discussed next. The interface uses low-overhead decode logic that translates loads/stores to reserved addresses into `POP/PUSH` commands that transfer pointers and data.

### 3.4.3 Reconfigurable Data Cache (R-DCache)

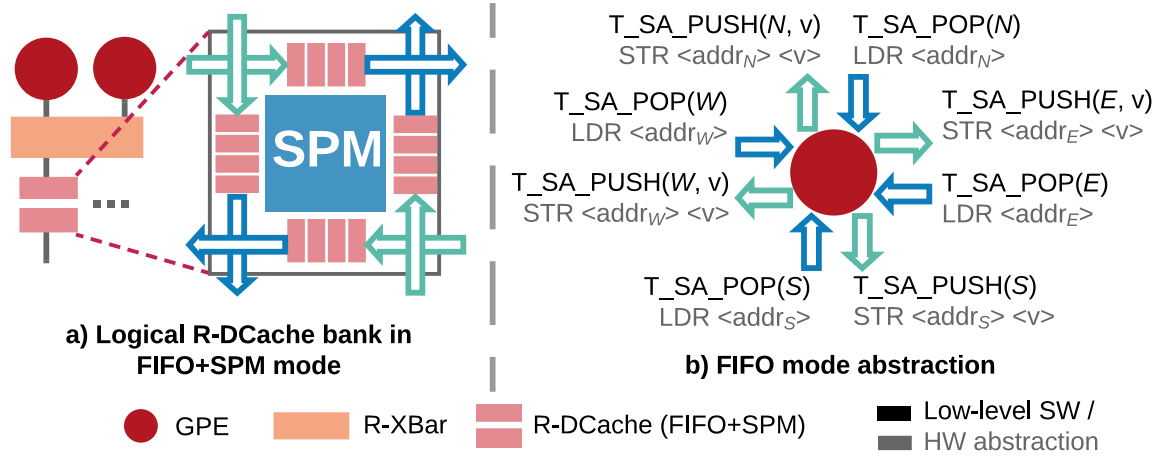


Figure 3.7: a) Logical view of an R-DCache bank in FIFO+SPM mode, with 4 FIFO partitions, one for each direction in 2D. b) Loads and stores to special addresses corresponding to each direction are mapped to POP and PUSH calls, respectively, into the FIFOs.

Transmuter has two layers of multi-banked memories, called reconfigurable data caches, *i.e.* R-DCaches (Fig. 3.6 – b, c). Each R-DCache bank is a standard cache module with enhancements to support the following modes of operation:

- **CACHE.** Each bank is accessed as a non-blocking, write-back, write-no-allocate cache with a least recently used replacement policy. The banks are interleaved at a set granularity, and a cacheline physically resides in one bank. Additionally, this mode uses a simple stride prefetcher to boost performance for regular kernels.
- **SPM.** The tag array, set-index logic, prefetcher, and MSHRs are powered off and the bank is accessed as a scratchpad.
- **FIFO+SPM.** A partition of the bank is configured as SPM, while the remainder is accessed as FIFO queues (Fig. 3.7 – left), using a set of head/tail pointers. The queue depth can be reconfigured using memory-mapped registers. The low-level abstractions for accessing the FIFOs are shown in Fig. 3.7 (right). This mode is used to implement spatial dataflow in Trans-SA (Fig. 3.5). This mode inherits the `ldr/str`-based PUSH/POP capabilities (Sec. 3.4.2).

### 3.4.4 Reconfigurable Crossbar (R-XBar)

A multicasting  $N_{src} \times N_{dst}$  crossbar creates one-to-one or one-to-many connections between  $N_{src}$  source and  $N_{dst}$  destination ports. Transmuter employs Swizzle-Switch Net-

work (SSN)-based crossbars that support multicasting [168, 88]. These and other works [3] have shown that crossbars designs can scale better, up to radix-64, compared to other on-chip networks. We augment the crossbar design with a crosspoint control unit (XCU) that enables reconfiguration by programming the crosspoints. A block diagram of a reconfigurable crossbar (R-XBar) is shown in Fig. 3.6-d.

The R-XBars support the following modes of operation:

- **ARBITRATE.** Any source port can access any destination port, and contended accesses to the same port get serialized. Arbitration is done in a single cycle using a Least-Recently Granted (LRG) policy [168], while the serialization latency varies between 0 and  $(N_{\text{src}} - 1)$  cycles. This mode is used in Trans-SC.
- **TRANSPARENT.** A requester can only access its corresponding resource, *i.e.* the crosspoints within the crossbar are set to 0 or 1 (Fig. 3.6-d). Thus, the R-XBar is transparent and incurs *no* arbitration or serialization delay in this mode. Trans-PS (in L1 and L2) and Trans-SA (in L2) employ TRANSPARENT R-XBars.
- **ROTATE.** The R-XBar cycles through a set of one-to-one port connections programmed into the crosspoints. This mode also has no crossbar arbitration cost. Fig. 3.8 illustrates how port multiplexing is used to emulate spatial dataflow in a 1D systolic array configuration (Trans-SA).

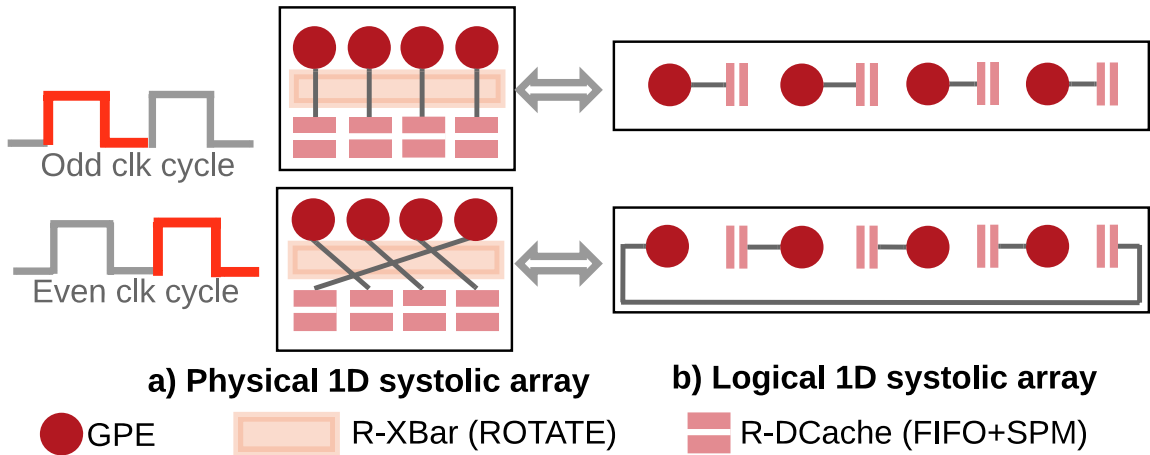


Figure 3.8: a) Physical and b) logical views of 1D systolic array connections within a Transmuter tile. Spatial dataflow is achieved by the R-XBar rotating between the two port-connection patterns.

There are two L1 R-XBars within a tile (Fig. 3.6-c). The upper R-XBar enables GPEs to access the L1 R-DCache, and the lower R-XBar amplifies the on-chip bandwidth between

the L1 and L2. Each L1 and L2 R-XBar has an additional “bypass” port to allow GPEs to communicate to the main memory when the corresponding R-DCache banks operate as SPMs. The L1 crossbars in the proposed Transmuter design have a bus width of 64 bits (32 address + 32 data bits) in each direction. L2 crossbars are wider (32 address + 128 data bits = 160 bits), as they transfer cachelines in bursts.

### 3.4.5 Synchronization Scratchpad Memory

Transmuter implements synchronization and enforces happens-before ordering using two approaches. The first is *implicit*, in the form of work/status/R-DCache queue accesses that block when the queue is empty or full. Second, it also supports *explicit* synchronization through a global synchronization SPM for programs that require mutexes, condition variables, barriers, and semaphores. For instance, say that GPEs 0 and 1 are to execute a Critical Section (CS) in a program. With explicit synchronization, the programmer can instantiate a mutex in the synchronization SPM and protect the CS with it. The same can also be achieved through implicit synchronization, with the following sequence of events: ① both GPEs  $\leftarrow$  LCP, ② LCP  $\rightarrow$  GPE0, ③ GPE0 executes the CS, ④ GPEs0  $\rightarrow$  LCP, ⑤ LCP  $\rightarrow$  GPE1, ⑥ GPE1 executes the CS, ⑦ GPE1  $\rightarrow$  LCP, where  $\leftarrow$  denotes POP-from and  $\rightarrow$  is a PUSH-to the work or status queue.

Compared to traditional hardware coherence, these techniques reduce power through lower on-chip traffic [99, 151]. Transmuter has a small global SPM, the Synchronization SPM, exclusively for synchronization operations that allow for the implementation of software coherence and standard primitives such as locks, condition variables, barriers, and semaphores. The synchronization SPM is interfaced to the LCPs and GPEs through a low-throughput two-level arbiter tree, as accesses to this SPM were not bottleneck for any of the evaluated workloads.

## 3.5 Transmuter Reconfiguration Design

Transmuter can self-reconfigure at run-time (initiated by an LCP) if the target configuration is known *a priori*. Reconfiguration can also be initiated by the host using a command packet with relevant metadata. The programming interface used to initiate such reconfiguration is discussed in Sec. 3.6.

Transmuter supports a MIMD paradigm with different cores running independent code, useful for applications composed of independent kernels, such as for wireless communication [127]. In order to support the MIMD paradigm, Transmuter consists of private

instruction caches for each GPE and LCP. The ICaches share access to the main memory with the L2 datapath through a two-level arbiter tree, not shown in Fig. 3.6. The GPE LS unit is augmented with logic to route packets to the work/status queue, synchronization SPM, and the L1 or L2 R-DCache, based on a set of base/bound registers. Reconfiguration changes the active base/bound registers, without external memory traffic. LCPs include similar logic but do not have access to the L1 or L2. Lastly, the system enables power-gating individual blocks, *i.e.* cores, R-XBars, and R-DCaches, based on reconfiguration messages. This is used to boost energy efficiency for memory-bound kernels.

Each step of the hardware reconfiguration happens in parallel and is outlined below.

- **GPE.** Upon receiving the reconfiguration command, GPEs switch the base/bound registers that their LS units are connected to (Sec. 3.5) in a single cycle.
- **R-XBar.** ARBITRATE  $\leftrightarrow$  TRANSPARENT reconfiguration entails a 1-cycle latency, as it only switches MUXes in the R-XBar (Fig. 3.6-d). The ROTATE mode uses set/unset patterns, which require a serial transfer of bit vectors from on-chip registers (*e.g.* a  $64 \times 64$  design incurs a 6-cycle latency<sup>2</sup>).
- **R-DCache.** Switching from CACHE to SPM mode involves a 1-cycle toggle of the scratchpad controller. The FIFO+SPM mode involves programming the head and tail pointer for each logical FIFO queue, which are transferred from control registers (4 cycles for 4 FIFO partitions).

Thus, the net reconfiguration time, accounting for buffering delays, amounts to  $\sim 10$  cycles, which is faster than FPGAs and many CGRAs (Sec. 3.2.1). For host-initiated reconfiguration, overheads associated with host-to-Transmuter communication leads to a net reconfiguration time of few 10s of cycles. We limit our discussions to self-reconfiguration in this work. Since Transmuter does not implement hardware coherence, switching between certain Transmuter configurations entails cache flushes from L1 to L2, from L2 to HBM, or both. The levels that use the SPM or FIFO+SPM mode do not need flushing. Furthermore, our write-no-allocate caches circumvent flushing for streaming workloads that write output data only once. Even when cache flushes are inevitable, the overhead is small (<1% of execution time) for the evaluated kernels in Sec. 3.9.

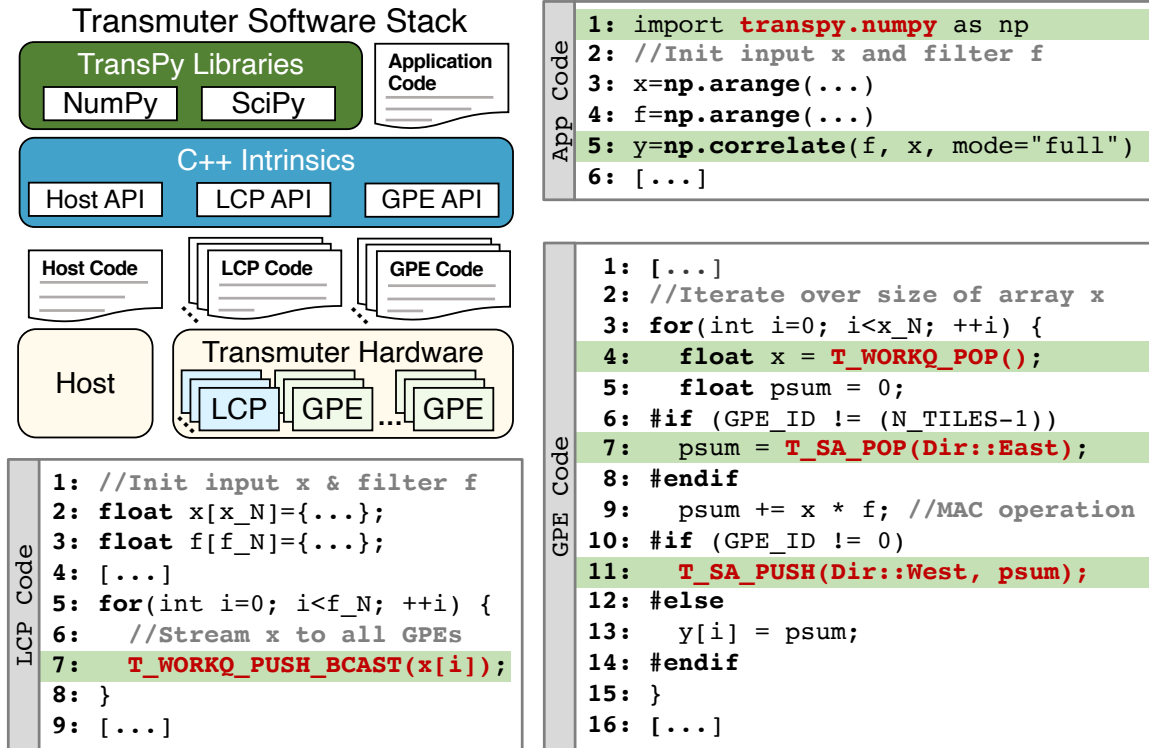


Figure 3.9: Transmuter software stack. Application code is written using Python and invokes library code for the host, LCPs, and GPEs. The implementations are written by experts using our C++ intrinsics library. Also shown is an example of a correlation kernel on Trans-SA (host library code not shown). The end-user writes standard NumPy code and changes only the import package to `transpy.numpy` (App:L1). Upon a library call (App:L5), the host performs data transfers and starts execution on Transmuter. The LCP broadcasts the vector  $x$  to all GPEs (LCP:L7). Each GPE pops the value (GPE:L4), performs a MAC using its filter value ( $f$ ) and east neighbor’s partial sum (GPE:L7), and sends its partial sum westward (GPE:L11). The last GPE stores the result into HBM. The host returns control to the application after copying back the result vector  $y$ .

### 3.6 Prototype Software Stack

We implement a software stack for Transmuter in order to support good programmability and ease of adoption of our solution. The software stack has several components: a high-level Python API, and lower-level C++ APIs for the host, LCPs, and GPEs. An outline of the software stack and a working Transmuter code example are shown in Fig. 3.9.

The highest level API, called TransPy, is a drop-in replacement for the well-known high-performance Python library NumPy, *i.e.* the TransPy API *exactly* mirrors that of NumPy. In the code example in Fig. 3.9, note that only one change is needed to convert

<sup>2</sup>Latency (in cycles) =  $\text{ceil}(N_{\text{rotate\_patterns}} \times N_{\text{dst}} \times \log_2(N_{\text{src}}) / \text{xfer\_width})$

the NumPy program to TransPy, *i.e.* the change of the imported module. Use of unsupported NumPy kernels will not result in an error - TransPy will simply fall back to the NumPy implementation. The `np.correlate` function is trapped in TransPy, dispatched to the Transmuter host layer, and a pre-compiled kernel library is invoked. We use `pybind11` [87] as the abstraction layer between Python and C++. TransPy also contains drop-in replacements for SciPy, PyTorch, NetworkX, and other libraries used in scientific computing, ML, graph analytics, *etc.*

Table 3.2: Critical host- and Transmuter-side C++ intrinsics used to write optimized kernel libraries (TID = Tile ID, GID = GPE ID). Note that the API is depicted for a single-cluster design, for simplicity.

Host-side Intrinsic Signature	Description
H_INIT ()	Initialize host-Transmuter interface
H_LAUNCH ()	Trigger Transmuter to start executing the kernel
H_FINISH ()	Wait (block) until Transmuter finishes executing
H_SEND_DATA (&dst, &src, size)	Mem-copy from external DRAM to HBM
H_RETR_DATA (&dst, &src, size)	Mem-copy from HBM to external DRAM
H_SET_†_ARG (argID, &arg, TID, [GID])	Copy an argument to an LCP or GPE
H_COMPILE_BIN (path_to_bin, flags)	Dynamically compile GPE/LCP code
H_LD_BIN_† (&bin, TID, [GID])	Stream compiled GPE/LCP binary into the HBM
H_SYNC_ALL ()	Synchronize with all LCPs and GPEs
H_RECONF (en_flag, TID, [GID])	Dynamically enable/disable GPE/LCP
H_RECONF<level> (config)	Trigger R-DCache/XBar reconfiguration
H_CLEANUP ()	Teardown host interface and deallocate structures
Transmuter-side Intrinsic Signature	Description
T_LD_WORD (addr)	Read a word from SPM; addr determines the bank
T_ST_WORD (addr, val)	Write a word into SPM; addr determines the bank
T_SA_POP (direction)	Pop data from systolic neighbor GPE
T_SA_PUSH (direction, val)	Push data to systolic neighbor GPE
T_+Q_PUSH (val, [GID])	Push data to work/status queue
T_+Q_POP ([GID])	Pop data from work/status queue
T_FREE_WORKQ_PUSH (val)	Push to the work queue of a free GPE
T_WORKQ_PUSH_BCAST (val)	Broadcast to all work queues in the tile
T_FLUSH<level> (bank)	Flush dirty data from to the next level
T_SPM_BOT<level, config> ()	Get a pointer to bottom of R-DCache/Sync. SPM
T_SPM_TOP<level, config> ()	Get a pointer to top of R-DCache/Sync. SPM
T_SYNC_LCPs ()	Synchronize with all LCPs in Transmuter
T_SYNC_TILE ()	Synchronize with all GPEs and LCP in the tile
T_SYNC_ALL ()	Synchronize with all LCPs, GPEs and host
T_SLEEP ()	Put self into sleep to conserve power
T_RECONF<level> (self_flag, config)	Self-reconfigure R-DCache/XBar / wait for host

†: LCP/GPE    +: WORK/STATUS

TransPy invokes kernels that are implemented by library writers and expert programmers, with the aid of the C++ intrinsics layer. A Transmuter SPMD kernel implementation

consists of three programs, one each for the host, LCP, and GPE. The host code is written in the style of OpenCL [179], handling data transfers to and from Transmuter, launching computation, initializing reconfigurable parameters (*e.g.* R-DCache FIFO depth), and triggering reconfiguration if needed. On the Transmuter side, notable API methods include those associated with the queue interface, for accessing SPMs and FIFOs, triggering cache flushes, and reconfiguration. Synchronization is handled using intrinsics that wrap around Portable Operating System Interface (POSIX) threads functions [141]. These calls allow for synchronization at different granularities, such as globally, within tiles, and across LCPs. A set of these intrinsics is listed in Table 3.2, and the code example in Fig. 3.9 reflects the use of some of these calls.

Thus, the Transmuter software stack is designed to enable efficient use of the Transmuter hardware by end-users, *without* the burden of reconfiguration and other architectural considerations. At the same time, the C++ layer allows expert programmers to write their own implementations, such as sophisticated heterogeneous implementations that partition the work between the host CPU and Transmuter. As an alternative to writing hand-tuned kernels for Transmuter, we are actively working on prototyping a compiler to automatically generate optimized C++-level library code for Transmuter based on the LIFT data-parallel language [178], the details of which are left for future work.

## 3.7 Experimental Methodology

This section describes the methodology used to derive performance, power, and area estimates for Transmuter. Table 3.3 shows the parameters used for modeling Transmuter. We compare Transmuter with an Intel Core i7 CPU and NVIDIA Tesla V100 GPU running optimized commercial libraries. The baseline specifications and libraries are listed in Table 3.4. For fair comparisons, we evaluate two different Transmuter designs, namely **TransX1** and **TransX8**, that are each comparable in area to the CPU and GPU, respectively. **TransX1** has a single  $64 \times 64$  Transmuter cluster and **TransX8** employs 8 such clusters. Both designs have one HBM2 stack/cluster to provide sufficient bandwidth.

### 3.7.1 Performance Models

We used the `gem5` simulator [20, 21] to model the Transmuter hardware. We modeled the timing for GPEs and LCPs after an in-order Arm Cortex-M4F, and cache and crossbar latencies based on a prior chip prototype that uses SSN crossbars [153, 156]. Data transfer/set-up times are excluded for all platforms. The average number of instructions



Table 3.3: Microarchitectural parameters of Transmuter gem5 model.

Module	Microarchitectural Parameters
GPE/LCP	1-issue, 4-stage, in-order (MinorCPU) core @ 1.0 GHz, tournament branch predictor, Functional Units (FUs): 2 integer (3 cycles), 1 integer multiply (3 cycles), 1 integer divide (9 cycles, non-pipelined), 1 FP (3 cycles), 1 LS (1 cycle)
Work/Status Queue	4 B, 4-entry FIFO buffer between each GPE and LCP within a tile, blocks loads if empty and stores if full
R-DCache (per bank)	CACHE: 4 kB, 4-way set-associative, 1-ported, non-coherent cache with 8 MSRs and 64 B block size, stride prefetcher of degree 2, word-granular (L1) / cacheline-granular (L2) SPM: 4 kB, 1-ported, physically-addressed, word-granular FIFO+SPM: 4 kB, 1-ported, physically-addressed, 32-bit head and tail pointer registers
R-XBar	$N_{src} \times N_{dst}$ non-coherent crossbar with 1-cycle response ARBITRATE: 1-cycle arbitration latency, 0 to $(N_{src}-1)$ serialization latency depending upon the number of conflicts TRANSPARENT: no arbitration, direct access ROTATE: switch port config. at programmable intervals Width: 32 address + 32 (L1) / 128 (L2) data bits
GPE/LCP ICache	4 kB, 4-way set-associative, 1-ported, non-coherent cache with 8 MSRs and 64 B block size
Sync. SPM	4 kB, 1-ported, physically-addressed scratchpad
Main Memory	1 HBM2 stack: 16 64-bit pseudo-channels, each @ 8000 MB/s, 80-150 ns average access latency

simulated was 2.3 billion (maximum 38.7 billion). Throughput is reported in FLOPS/s and only accounts for useful (algorithmic) FLOPS.

The resource requirement for simulations using this detailed gem5 model is only tractable for Transmuter systems up to  $8 \times 16$ . For larger systems, we substitute the gem5 cores with trace replay engines while retaining the gem5 model for the rest of the system. Offline traces are generated on a native machine and streamed through these engines. This allows us to simulate systems up to one  $64 \times 64$  cluster. On average, across the evaluated kernels, the trace-driven model is pessimistic to 4.5% of the execution-driven model. For a multi-cluster system, we use analytical models from gem5-derived bandwidth and throughput scaling data (Sec. 3.9.5).

We implemented each kernel in C++ and hand-optimized it for each Transmuter configuration using the intrinsics discussed in Sec. 3.6. Compilation was done using an Arm GNU compiler with the `-O2` flag. Each GPE and LCP executes its own process and the processes communicate through loads and stores to a set of shared pages in the global address space. All experiments used single-precision FP arithmetic.

Table 3.4: Specifications of baseline platforms and libraries evaluated.

Platform	Specifications	Library Name and Version
CPU	Intel i7-6700K, 4 cores/8 threads at 4.0-4.2 GHz, 16 GB DDR3 memory @ 34.1 GB/s, AVX2, SSE4.2, 122 mm <sup>2</sup> (14 nm)	MKL 2018.3.222 (GEMM / GEMV / SpMM / SpMV), DNNL 1.1.0 (convolution), FFTW 3.0 (FFT)
GPU	NVIDIA Tesla V100, 5120 CUDA cores at 1.25 GHz, 16 GB HBM2 memory at 900 GB/s, 815 mm <sup>2</sup> (12 nm)	cuBLAS v10 (GEMM/GEMV), cuDNN v7.6.5 (convolution), cuFFT v10.0 (FFT), CUSP v0.5.1 (SpMM), cuSPARSE v8.0 (SpMV)

### 3.7.2 Power and Area Models

To obtain accurate estimations for power and area, we designed Register-Transfer Level (RTL) models for Transmuter hardware blocks and synthesized them. The GPEs and LCPs are modeled as Arm Cortex-M4F cores. For the R-XBar, we use the SSN design proposed in [168], augmented with an XCU. The R-DCaches are cache modules enhanced with SPM and FIFO control logic. The crossbar and core power models are based on RTL synthesis reports and the Arm Cortex-M4F specification document. The R-XBar power model is calculated based on the values obtained from synthesis results and calibrated against the data reported in [168]. For the caches and synchronization SPM, we used CACTI 7.0 [15] to estimate the dynamic energy and leakage power. We further verified our power estimate for SpMM on Transmuter against a prior SpMM ASIC prototype [153] and obtained a pessimistic deviation of 17% after accounting for the architectural differences. Finally, the area model uses estimates from synthesized Transmuter blocks and is also cross-verified with the simplified 40 nm chip prototype.

We note that this work considers only the chip power on all platforms, for fair comparisons. We used standard profiling tools for the CPU and GPU, namely `nvprof` and `RAPL`. For the GPU, we estimated the HBM power based on per-access energy [150] and the measured memory bandwidth and subtracted it out. The power is scaled for iso-technology comparisons using quadratic scaling.

## 3.8 Kernel Implementations on Transmuter

Transmuter is built using COTS cores that lend the architecture to be kernel-agnostic. Here, we present our mappings of the fundamental kernels in Sec. 3.2 on the selected Transmuter configurations, and list the code snippets for three of our implementations. Then, we analyze the performance of each kernel on the selected Transmuter configurations. Additional kernels in the domain of linear algebra have been mapped and evaluated on a preliminary version of Transmuter for different resource sharing configurations [173]. It is worth noting

that while executing memory-bound kernels, Transmuter can power-down resources within a tile to conserve energy when the available memory bandwidth is saturated, without hurting the system throughput.

### 3.8.1 Dense Matrix Multiplication and Convolution

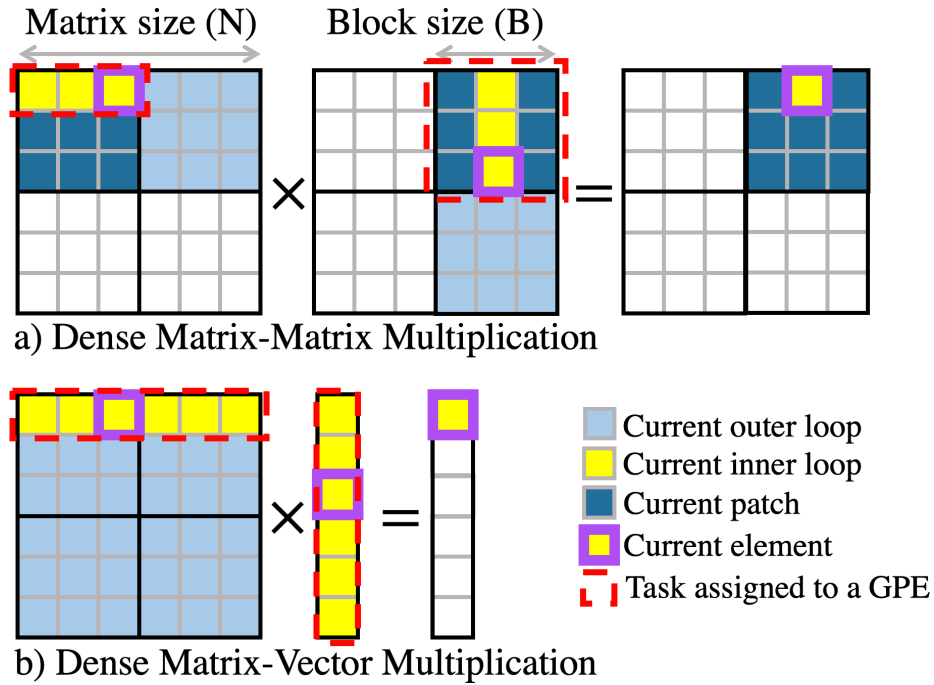


Figure 3.10: Illustration of dense matrix-matrix and matrix-vector multiplication kernels mapped onto Transmuter.

We describe the mappings of two common Level-2 and Level-3 Basic Linear Algebra Subprograms (BLAS) kernels, namely GEMM and GEMV multiplication.

**GEMM.** GEMM is a compute-bound kernel that performs  $C = \alpha \cdot A \cdot B + \beta \cdot C$ , where  $\alpha$  and  $\beta$  are scalars, while  $A$ ,  $B$ , and  $C$  are dense matrices. GEMM produces  $O(N^3)$  FLOPS for  $O(N^2)$  fetches and exhibits very high reuse [74]. It also presents contiguous accesses, thus showing amenability to a shared memory based architecture. Without loss of generality, GEMM is evaluated on Transmuter with  $N \times N$  square matrices.

Our implementation of GEMM on Trans-SC uses a common blocking optimization [125] (Fig. 3.10-a). Based on our discussion in Sec. 3.2.3, Trans-SC is a good fit for blocked-GEMM, as the L1 and L2 shared caches exploit this reuse efficiently. We similarly implement GEMM on Trans-PS but with the blocked partial results stored in the private L1 SPMs. Naturally, Trans-PS misses the opportunity for data sharing. For Trans-SA, the

GPEs execute GEMM in a systolic fashion with the rows of  $A$  streamed through the L2 cache, and the columns of  $B$  loaded from the L1 SPM.

**GEMV.** GEMV is defined by  $y = \alpha \cdot A \cdot x + \beta \cdot y$ , where  $\alpha$  and  $\beta$  are scalars,  $x$  and  $y$  are vectors and  $A$  is a dense matrix. It is a memory-bound kernel that involves lower FLOPS/B —  $O(N^2)$  FLOPS for  $O(N^2)$  fetches — than GEMM, but still involves contiguous memory accesses [55]. The Trans-SC and Trans-PS implementations are similar to those for GEMM, but blocking is not implemented due to lower data reuse (Fig. 3.10-b). Each GPE performs a dot-product of an  $A$ -row with  $x$ , along with the element-wise operations. On Trans-SA, the vector is streamed into each GPE through the L2 cache, while the matrix elements are fetched from the L1 SPM. Each GPE performs a Multiply-And-Accumulate (MAC) and passes the partial sum and input matrix values to its neighbors. We avoid network deadlock in our GEMM and GEMV Trans-SA implementations by reconfiguring the FIFO depth of the L1 R-DCache (Sec. 3.4.3) to allow for sufficient buffering. A pseudocode snippet for GEMV Trans-SC implementation is shown in Program 3.1, and the API definitions are in Table 3.2.

```
void GEMV_LCP(int start, int end, int N_G) {
    // start: start row index, end: end row index
    // N_G: number of GPEs per tile
    for (int row = start; row < end; row++) {
        T_WORKQ_PUSH(gid, row);
        gid = (gid == N_G - 1) ? 0 : (gid + 1);
    } // gid is GPE ID
    T_WORKQ_PUSH_BCAST(-1);
}
void GEMV_GPE(Matrix A, Vector x, Vector y, N, int a, int b) {
    // y = a * A * x + b * y
    // N: matrix and vector dimension
    int row = 0;
    while ((row = T_WORKQ_POP()) != -1) {
        float psum = 0.0;
        for (int col = 0; col < N; col++) {
            psum += A[row][col] * x[col]
        }
        y[row] = b * y[row] + a * psum;
    }
}
```

Program 3.1: GEMV pseudocode on Transmuter in Trans-SC.

**Convolution.** Convolution in 2D is comprised of adding each input element (*e.g.* image

pixels) to its neighbors, weighted by a filter, to obtain an output element. Convolution produces  $(2 \cdot F^2 \cdot N^2 \cdot IC \cdot OC)/S$  FLOPS, for an  $F \times F$  filter convolving with stride  $S$  over an  $N \times N$  image, with  $IC$  input and  $OC$  output channels. The filter is reused while computing one output channel, and across multiple images. Input reuse is limited to  $O(F \cdot OC)$ , for  $S < F$ . On Trans-SC, we assign each GPE to compute the output of multiple rows, to maximize the filter reuse across GPEs. For Trans-PS and Trans-SA, we statically partition each image into  $B \times B \times IC$  sub-blocks, such that the input block and filter fit in the private L1 SPM. Each block is then mapped to a GPE for Trans-PS, and to a set of  $F$  adjacent GPEs of a 1D systolic array for Trans-SA using a row stationary approach similar to [31].

### 3.8.2 Fast Fourier Transform

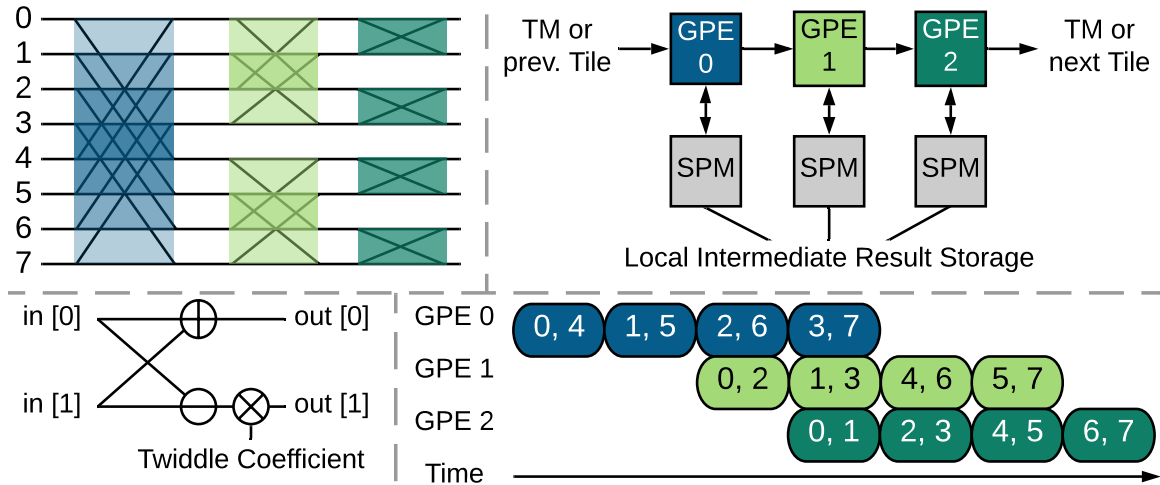


Figure 3.11: Top: Mapping of FFT stages onto GPEs in Transmuter (Trans-SA). Bottom: Each GPE executes butterfly operations greedily, leading to a fully-pipelined schedule.

**FFT.** FFT in 1D computes an  $N$ -point discrete Fourier transform in  $\log(N)$  sequential *stages*. Each stage consists of  $N/2$  *butterfly* operations. While an entire stage can be computed in parallel if the input data is all available, FFT applications often operate on streaming input samples, where operands are produced at a constant rate. Thus, FFT is amenable to spatial dataflow architectures [86, 47]. Our Trans-SA mapping is similar to pipelined systolic ASICs; each stage is assigned to a single GPE, and each GPE immediately pushes its outputs to its neighbor. The butterflies in each stage are computed greedily. This is illustrated in Fig. 3.11. To reduce storage and increase parallelism, Trans-SA uses run-time twiddle coefficient generation when the transform size is too large for on-chip memory, *e.g.*  $>256$  for  $2 \times 8$ , with the trade-off of making the problem compute-bound. For sizes larger

than  $2^{N_{GPEs}}$ , FFT computation needs to be time-multiplexed across GPEs. For smooth cross-tile transfers, the L2 in this mapping is also configured as 1D systolic, thus precluding corner case handling in software. A pseudocode snippet for LCP and GPE for FFT Trans-SA implementation is shown in Program 3.2, and the API definitions are in Table 3.2. On Trans-SC, the butterfly operations are distributed evenly among GPEs to compute a stage in parallel. LCPs assign inputs and collect outputs from GPEs. All cores synchronize after each stage. For Trans-PS, the same scheduling is used and partial results are stored in the L1 SPM.

### 3.8.3 Sparse Matrix Multiplication

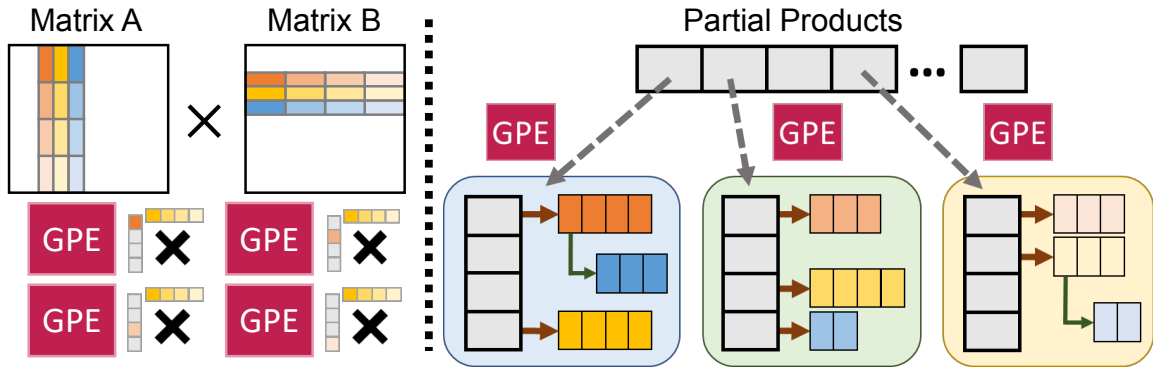


Figure 3.12: SpMM mapping on Transmuter. Left: Multiply phase: each GPE multiplies an element of a column of  $A$  with a row of  $B$ , generating a partial product matrix. Right: Merge phase: each GPE independently streams in partial product matrix rows, performs mergesort, and stores the result.

**SpMM.** SpMM is a memory-bound kernel with low FLOPS that decrease with increasing sparsity, *e.g.*  $\sim 2N^3r_M^2$ , for uniform-random  $N \times N$  matrices with density  $r_M$ . Furthermore, sparse storage formats lead to indirection and thus irregular memory accesses [118, 151]. We implement SpMM in Trans-SC using a prior outer product approach [151] (Fig. 3.12). In the *multiply phase* of the algorithm, the GPEs multiply a column of  $A$  with the corresponding row of  $B$ , such that the row elements are reused in the L1 cache. In order to exploit this reuse, both the L1 and L2 layers of Transmuter are configured as shared caches. In the *merge phase*, a GPE merges all the partial products corresponding to one row of  $C$ . Each GPE maintains a private list of sorted partial results and fills it with data fetched from off-chip. Trans-PS operates similarly, but with the sorting list placed in private L1 SPM, given that SPMs are a better fit for operations on disjoint memory chunks. This prevents cache pollution that would have occurred while operating

```

void FFT_LCP(Complex* input, Complex* output, int N,
             bool is_input, bool is_output) {
    // N: FFT size, log2(N): number of FFT stages
    if (is_input) {
        for (int i = 0; i < N; i++) {
            T_WORKQ_PUSH(0, input[i]);
        }
    }
    if (is_output) {
        for (int i = 0; i < N; i++) {
            output[i] = T_STATUSQ_POP(log2(N)-1);
        }
    }
}

void FFT_GPE(Complex* input, Complex* output, int N,
             int N_G, int S, int P) {
    // N_G: number of GPEs per tile
    // S: step size, P: next step size
    int id = gid + tid * N_G; // gid: GPE ID, tid: Tile ID
    Complex *sp = T_SPM_BOT<Lev::L1, Conf::systolic_array_ld>();
    for (int i = 0; i < N/2; i++) {
        in1= (id == 0) ? T_WORKQ_POP() : T_SA_POP(Dir::West);
        in2= (id == 0) ? T_WORKQ_POP() : T_SA_POP(Dir::West);
        out1,out2 = compute_butterfly(in1,in2);
        if (id == log2(N) - 1)
            T_STATUSQ_PUSH(out1); T_STATUSQ_PUSH(out2);
        else {
            T_ST_WORD(sp + i, out1); T_ST_WORD(sp + i + S, out2);
            if (i > P - 1) {
                T_SA_PUSH(Dir::East, T_LD_WORD(sp + i - P));
                T_SA_PUSH(Dir::East, out1);
            }
        }
    }
    for (int i = 0; i < P; i++) {
        T_SA_PUSH(Dir::East, T_LD_WORD(sp + S + i));
        T_SA_PUSH(Dir::East, T_LD_WORD(sp + S + P + i));
    }
}

```

Program 3.2: FFT pseudocode on Transmuter in Trans-SA.

in Trans-SC, but trades-off with a loss of reuse during the multiply phase. Lastly, SpMM in Trans-SA is implemented following a recent work that uses sparse packing [79]. Directly

mapping SpMM on Transmuter in systolic mode would lead to degraded GPE utilization. A recent study proposes to map sparse matrices onto a systolic array for acceleration [111]. By packing the sparse matrices into their dense counterpart, the utilization of the systolic array running SpMM could be improved significantly. The computation is equally split across the tiles.

```

void SpMV_LCP() {
    T_WORKQ_PUSH_BCAST(1);
}
void SpMV_GPE(A_row, A_col, A_val, A_part, x, x_part, y, N, P) {
    // y = A * x, gid: GPE ID, tid: Tile ID
    // P: row partition per tile, N: matrix/vector dimension
    // N_T: number of tiles, N_G: number of GPEs per tile
    T_WORKQ_POP();
    int parts_per_tile = ceil(N/N_T);
    int i = A_part[gid * N_T * N_G + tid];
    for (int part = tid * parts_per_tile;
        part < (tid + 1) * parts_per_tile; part++) {
        int b_start = x_part[part][gid];
        int b_end = x_part[part][gid + 1];
        float *sp = T_SPM_BOT<Lev::L1, Conf::systolic_array_ld>();
        float *sp_start = sp;
        for (int j = b_start; j < b_end; j++)
            T_ST_WORD(sp++, x[j]);
        float *sp_sum = sp;
        for (int row = part * P; row < (part + 1) * P; row++) {
            float psum = 0;
            while (A_row[i] == row)
                psum += A_val[i++] * T_LD_WORD(sp_start + A_col[i]);
            T_ST_WORD(sp++, psum);
        }
        for (int row = 0; row < P; row++) {
            float popped = (gid == 0) ? T_SA_POP(Dir::West) : 0;
            float sum = popped + T_LD_WORD(sp_sum + row);
            if (gid == N_G - 1) y[part * P + row] = sum;
            else T_SA_PUSH(Dir::East, sum);
        }
    }
}

```

Program 3.3: SpMV pseudocode on Transmuter in Trans-SA.

**SpMV.** SpMV, similar to SpMM, is bandwidth-bound and produces low FLOPS ( $\sim$



$2N^2r_Mr_v$  for a uniformly random  $N \times N$  matrix with density  $r_M$ , and vector with density  $r_v$ ). We exploit the low memory traffic in the outer product algorithm for sparse vectors, mapping it to Trans-SC and Trans-PS. The vector NZEs are distributed across the GPEs, so each GPE generates partial product vectors. Rather than writing the partial products to memory, the GPEs send them to the LCP through the status queues. Each LCP performs the partial *merge* phase on the partial products of its own tile that are streamed in through the work queue. Finally, the partially merged vectors are sent to *Tile 0* through the synchronization SPM and the LCP in *Tile 0* performs the final merge across the partial vectors generated by each tile and writes the resulting vector to memory. SpMV on 1D Trans-SA is implemented using inner product on a packed sparse matrix as described in [79]. The packing algorithm packs 64 rows as a slice and assigns one slice to each  $1 \times 4$  sub-tile within a tile. Each GPE loads the input vector elements into the SPM, fetches the matrix element, and performs MAC operations, with the partial results being streamed to its neighbor within the sub-tile. A pseudocode snippet for SpMV Trans-SA implementation is shown in Program 3.3, and the API definitions are in Table 3.2.

Finally, for both SpMM and SpMV, we use dynamic scheduling for work distribution to the GPEs (Sec. 3.4.1), in order to fully exploit the amenability of sparse workloads to SPMD architectures [151].

### 3.8.4 Performance with Different Configurations

Fig. 3.13 presents the comparisons between Trans-SC, Trans-PS, and Trans-SA in terms of performance. This analysis was done on a small  $2 \times 8$  system, such that the working sets spill out of on-chip memory and stress the hardware. The results show that the *best-performing Transmuter configuration is kernel-dependent*, and in certain cases *also input-dependent*. Fig. 3.14 shows the cycle breakdowns and the work imbalance across GPEs.

For *GEMM*, Trans-SC achieves high L1 hit rates ( $>99\%$ ), as efficient blocking leads to good data reuse. Trans-PS suffers from capacity misses due to lack of sharing, noted from the large fraction of L2 misses. Further, Trans-SC performs consistently better than Trans-SA, as it does not incur the overhead of manually fetching data into the L1 SPM. For *GEMV*, Trans-SC and Trans-PS behave the same as GEMM. However, Trans-SA experiences cache thrashing (increasing with matrix size) in the private L2. For *Convolution*, as with GEMM/GEMV, Trans-SC performs the best due to a regular access pattern with sufficient filter and input reuse. Across these kernels, stride prefetching in Trans-SC is sufficient to capture the regular access patterns.

For *FFT*, Trans-SA achieves significantly higher throughput because it benefits from

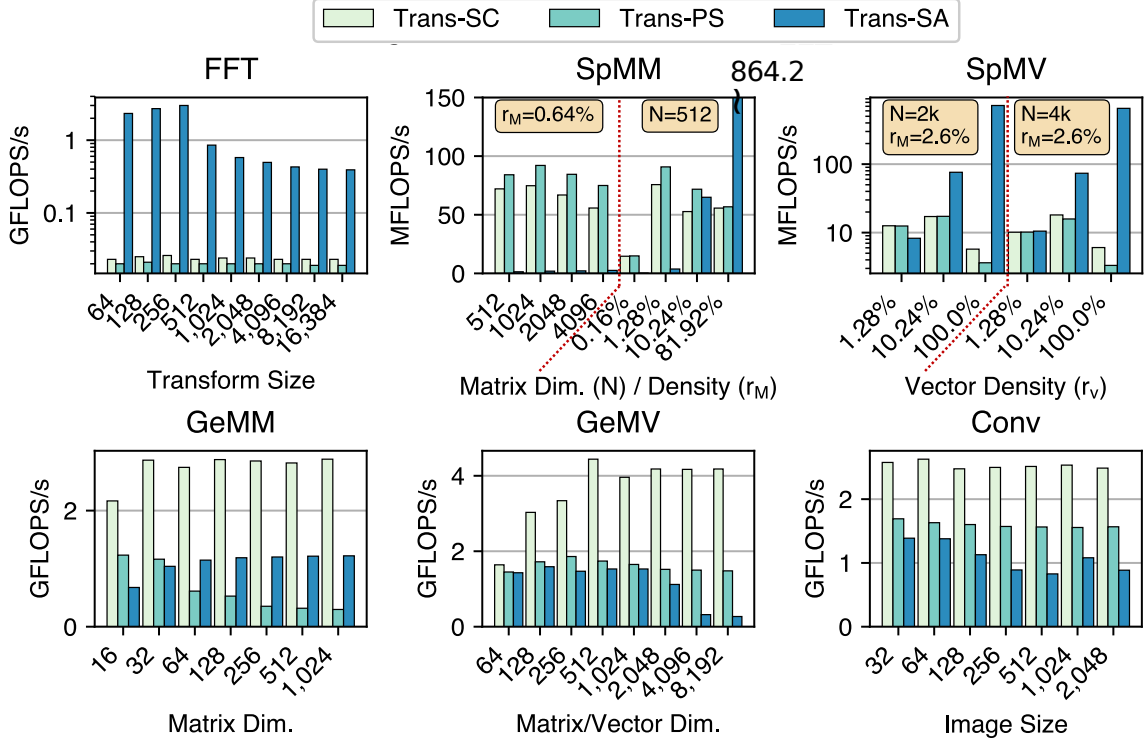


Figure 3.13: Performance of  $2 \times 8$  Trans-SC, Trans-PS, and Trans-SA configurations across different inputs for the kernels in Sec. 3.8. All matrix operations are performed on square matrices without loss of generality. Convolution uses  $3 \times 3$  filters, 2 input/output channels, and a batch size of 2.

the streaming inputs and exploits better data reuse, evidenced by  $\sim 10 \times$  less memory bandwidth usage compared to Trans-SC/Trans-PS. Trans-SA performs better for sizes  $< 512$  compared to other sizes, as the twiddle coefficients are loaded from on-chip rather than being computed. Despite the intra-stage parallelism, low reuse in Trans-PS results in  $> 25\%$  L1 misses. More importantly, inter-GPE synchronization and coherence handling at the end of each stage limit the performance for Trans-SC/Trans-PS. Finally, the control flow in the non-systolic code is branchy and contributes to expensive ICache misses.

For *SpMM*, the performance is highly dependent on input matrix dimension and sparsity. Trans-SC favors smaller/sparser matrices, whereas Trans-PS performs the best for larger matrices with moderate sparsity. The multiply phase of outer product is better suited to Trans-SC as the second input matrix rows are shared. The merge phase is amenable to Trans-PS since the private SPMs overcome the high thrashing that Trans-SC experiences while merging multiple disjoint lists. *SpMM* achieves an additional performance boost of 3.2% by reconfiguring between the two phases (not shown in Fig. 3.13). Trans-SA dominates for matrices with densities  $> \sim 11\%$ . However, it performs poorly in comparison to

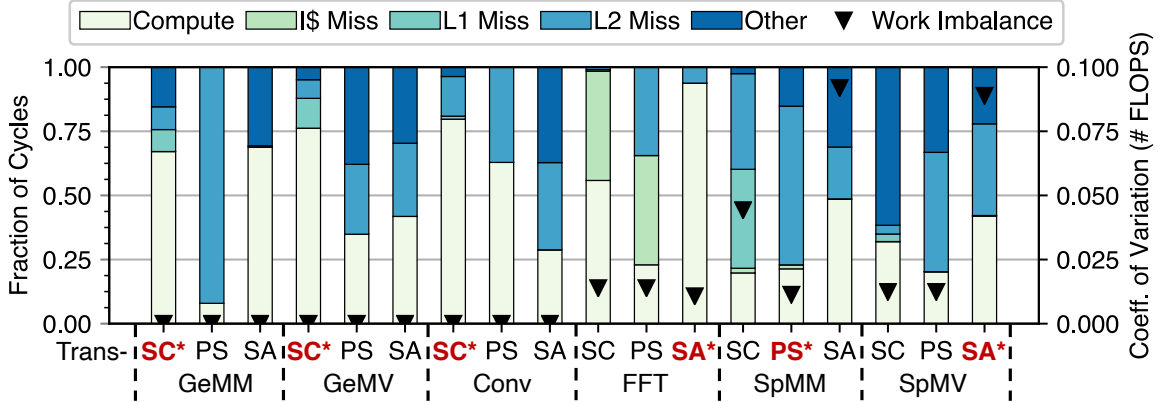


Figure 3.14: Cycle breakdown for the kernels in Sec. 3.8. \* (red) indicates the best-performing configuration. “Other” comprises stalls due to synchronization and bank conflicts. ▼: work imbalance across GPEs ( $\sigma/\mu$  of # FLOPS). Inputs are: 1k (GEMM), 8k (GEMV), 2k (Convolution), 16k (FFT), 4096, 0.64% (SpMM), 4k, 2.6%, dense vector (SpMV).

outer product for highly-sparse matrices. Although  $\sim 50\%$  of the time is spent on computations, as shown in Fig. 3.14, most of which are wasted on fetched data that are discarded after failed index matches.

For *SpMV*, the performance depends on the input matrix size, dimensions, as well as the vector density. Notably, Trans-SA benefits through the spatial dataflow for SpMV but not for SpMM, because the SpMV implementation treats the vector as dense, and thus can stream in the vector elements efficiently into the GPE arrays. At sufficiently high vector sparsities, outer product on Trans-SC/Trans-PS outperforms Trans-SA by avoiding fetches of zero-elements. However, for higher densities, Trans-SC and Trans-PS suffer from the overhead of performing mergesort that involves frequent GPE-LCP synchronization, and the serialization at LCP 0.

**Takeaways.** Demand-driven dataflow with shared caching outperforms other configurations for GEMM, GEMV, and convolution due to sufficient data sharing and reuse. Streaming kernels such as FFT and SpMV (with dense vectors) are amenable to spatial dataflow. SpMM and high-sparsity SpMV show amenability to private scratchpads or shared caches depending on the input size and sparsity, with the systolic mode outperforming only for very high densities.

## 3.9 Evaluation

In this section, we first compare the best-performing Transmuter kernel implementations to the CPU, the GPU, and existing FPGAs, CGRAs, and ASICs. Then we show the power and area analysis. Next, we deep-dive into the evaluation of an application that exercises rapid reconfiguration. Finally, we present further analysis of Transmuter on throughput and bandwidth, design space exploration, and control divergence and data reuse.

### 3.9.1 Comparison with the CPU and GPU

We now compare the best-performing Transmuter configuration with the CPU and GPU running optimized commercial libraries (Table 3.4). The throughput and energy-efficiency gains of Transmuter for each kernel in Sec. 3.8 are presented in Fig. 3.15. We compare TransX1 to the CPU and TransX8 to the GPU, as discussed in Sec. 3.7.

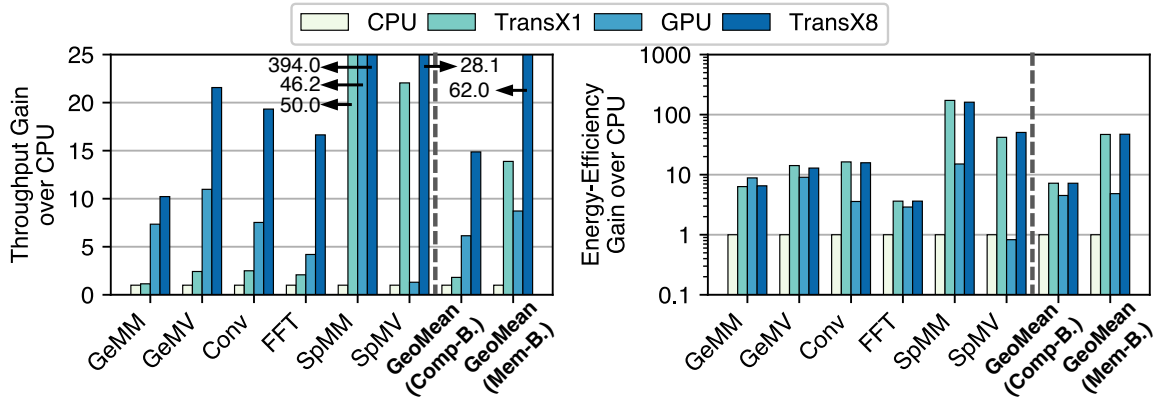


Figure 3.15: Throughput (left) and energy-efficiency (right) improvements of Transmuter over the CPU and GPU. Data is averaged across the inputs: 256-1k (GEMM), 2k-8k (GEMV), 512-2k (Convolution), 4k-16k (FFT), 1k-4k, 0.64% ( $r_M$ ), 10.2%-100% ( $r_v$ ) (SpMM). Geometric mean improvements for the compute-bound and memory-bound kernels are shown separately.

**Compute-Bound Kernels (GEMM, Convolution, FFT).** TransX1 harnesses high data-level parallelism, and thus achieves performance improvements of 1.2-2.5 $\times$  over the CPU, despite clocking at  $\frac{1}{4}$ th the speed of the deeply-pipelined CPU cores. The true benefit of Transmuter’s simple cores and efficient crossbars appears in the form of energy-efficiency gains, ranging from 3.6-16.3 $\times$ , which is owed largely to the high power consumption of the bulky out-of-order CPU cores. Over the GPU, TransX8 gets performance gains of 1.3-2.6 $\times$  and efficiency improvements of 0.8-4.4 $\times$  with an efficient implementation on Trans-SC for GEMM and convolution. The  $\sim 20\%$  energy-efficiency loss

for GEMM is explained by the amenability of GEMM to a Single Instruction, Multiple Threads (SIMT) paradigm; although the performance is similar between SIMT and SPMD, SPMD incurs slightly larger energy costs associated with higher control overhead over SIMT. On FFT, Transmuter sustains consistent performance scaling using the spatial dataflow of Trans-SA, with each tile operating on an independent input stream, thus leading to minimized conflicts. The gap between throughput gain ( $4.0\times$ ) and energy-efficiency gain ( $1.3\times$ ) over the GPU is explained by the `cuFFT` algorithm which is more efficient for batched FFTs.

**Memory-Bound Kernels (GEMV, SpMM, SpMV).** `TransX1` on GEMV achieves a  $2.4\times$  better throughput over the CPU, with the CPU becoming severely DRAM-bound ( $>98\%$  bandwidth utilization) for input dimensions beyond 1,024. The  $14.2\times$  energy-efficiency gain of `TransX1` stems from tuning down the number of active GPEs to curtail bandwidth starvation, thus saving power.

On SpMM and SpMV, the performance of Transmuter is highly sensitive to the densities and sizes of the inputs, with improvements ranging from  $4.4\text{-}110.8\times$  over the CPU and  $5.9\text{-}37.7\times$  over the GPU. With SpMM, execution in Trans-PS enables overcome the CPU’s limitation of an inflexible cache hierarchy, as well as harnesses high MLP across the sea of GPEs. While Transmuter is memory-bottlenecked for SpMM, SpMV is bounded by the scheduling granularity of the packing algorithm deployed on Trans-SA. Despite that, for SpMV, `TransX1` outperforms both the CPU as well as the GPU that has  $7.2\times$  greater available bandwidth. In the case of the GPU, while there are sufficient threads to saturate the Streaming Machines (SMs), the thread divergence in the SIMT model is the bottleneck. The GPU achieves just  $0.6\%$  and  $0.002\%$  of its peak performance, respectively for SpMM and SpMV, impaired by memory and synchronization stalls. In comparison, SPMD on Transmuter reduces synchronization, resulting in  $21\text{-}42\%$  time spent on useful computations (Fig. 3.14). For SpMM, the outer product implementation demonstrates ASIC-level performance gains of  $5.9\text{-}11.6\times$  [151] over the GPU, by minimizing off-chip traffic and exploiting the asynchronicity between GPEs. As with GEMV, disabling bandwidth-starved resources contributes to the energy-efficiency gains.

**Effect of Iso-CPU Bandwidth.** `TransX1` uses one HBM stack that provides 125 GB/s peak bandwidth, about  $3.6\times$  greater than the DDR3 bandwidth to the CPU. If given the bandwidth of the DDR3 memory, `TransX1` still achieves performance gains averaging  $17.4\times$  and  $6.3\times$  for SpMM and SpMV, respectively. For GEMV, `TransX1` remains within a modest  $6\text{-}8\%$  of the CPU with this low bandwidth.

### 3.9.2 Comparison with FPGA, CGRA, and ASIC

Table 3.5 shows the estimated energy-efficiency improvements of Transmuter over recent FPGA, CGRA, and ASIC implementations. The efficiencies reported in prior work are scaled quadratically for iso-technology comparisons with Transmuter. Overall, Transmuter achieves average efficiency gains of  $3.4\times$  and  $2.0\times$  over FPGAs and CGRAs, respectively, and is within  $9.3\times$  (maximum  $32.1\times$ ) of state-of-the-art ASICs for the evaluated kernels.

Table 3.5: Energy-efficiency improvements (black) and deteriorations (red) of Transmuter over prior FPGAs, CGRAs, and ASICs.

Platform	GEMM	GEMV	Convolution	FFT	SpMM	SpMV
FPGA	$2.7\times$ [65]	$8.1\times$ [115] <sup>3</sup>	$2.7\times$ [210]	$2.2\times$ [65]	$3.6\times$ [66]	$3.0\times$ [45]
CGRA	$2.2\times$ [162]	$3.0\times$ [36]	$1.2\times$ [36]	$1.0\times$ [95] <sup>4</sup>	$1.9\times$ [36]	$2.9\times$ [36]
ASIC	$(32.1\times)$ [157]	$(10.5\times)$ [167]	$(13.8\times)$ [189] $(7.6\times)$ [167]	$(18.1\times)$ [158] $(17.0\times)$ [49]	$(3.0\times)$ [153] $(4.1\times)$ [216]	$(3.9\times)$ [151]

<sup>3</sup>Performance/bandwidth used as power is N/A. <sup>4</sup>Estimated for floating-point based on [187].

### 3.9.3 Power and Area

Table 3.6 details the power consumption and area footprint of a  $64\times 64$  Transmuter cluster in 14 nm. Most of the power is consumed by the network and memory, *i.e.* L1 R-XBars, R-DCaches, and ICaches, while the cores only consume 20.8%. This is consistent with a growing awareness that the cost of computing has become cheaper than the cost to move data, even on-chip [81]. GPEs and L1 R-XBars, the most frequently switched modules, consume 84.2% of the total dynamic power. The estimated power for a single Transmuter cluster is 13.3 W in 14 nm with an area footprint within 1.7% of the CPU’s area. The estimated worst-case reconfiguration overhead is 74.9 nJ.

### 3.9.4 End-to-End Workload Analysis

We report the estimated speedups of Transmuter over the CPU and GPU for the end-to-end workloads (Fig. 3.3) in Table 3.7. File I/O and cross-platform (*e.g.* CPU→GPU) data transfer times are excluded for all platforms. Overall, Transmuter achieves speedups averaging  $3.1\times$  over the CPU and  $3.2\times$  over the GPU.

Next, we elucidate how rapid reconfiguration enables efficient execution of workloads that involve mixed sparse-dense computation in an inner loop. We make a case study on a representative mixed-data application, namely *Sinkhorn*, that performs iterative computation to determine the similarity between documents [114, 166]. Sinkhorn computation

Table 3.6: Power and area of a  $64 \times 64$  Transmuter cluster in 14 nm.

Module	Power (mW)			Area (mm <sup>2</sup> )
	Static	Dynamic	Total	
GPE Cores	361.3	2380.5	2741.7	28.9
LCP Cores	5.6	22.5	28.1	0.4
Sync. SPM	0.6	0.1	0.6	0.1
All ICaches	2566.6	373.6	2940.1	25.7
LCP DCaches	39.5	0.9	40.4	0.5
L1 R-DCaches	2527.1	204.0	2731.0	30.7
L2 R-DCaches	37.4	18.3	55.7	0.5
L1 R-XBars	1757.8	2149.3	3907.1	30.3
L2 R-XBars	36.9	14.8	51.7	0.8
MUXes/Arbiters	581.9	87.6	669.5	0.7
Memory Controllers	47.5	129.0	176.4	5.5
<b>Total</b>	<b>8.0 W</b>	<b>5.4 W</b>	<b>13.3 W</b>	<b>124.1 mm<sup>2</sup></b>

Table 3.7: Estimated speedups for the end-to-end workloads in Fig. 3.3.

Speedup	DANMF	LSTM	Marian	MaxCut	MFCC
TransX1 vs. CPU	4.1×	1.1×	2.2×	6.2×	1.7×
TransX8 vs. GPU	3.5×	3.8×	2.1×	7.2×	1.6×
	NBSGD	RolePred	SemSeg	Sinkhorn	VidSeg
TransX1 vs. CPU	3.5×	2.7×	2.4×	3.1×	2.2×
TransX8 vs. GPU	2.8×	2.3×	2.5×	3.0×	2.8×

typically involves large, sparse matrices in conjunction with dense matrices. We implement the algorithm described in [35] (Algorithm 3.9.1). The inner loop has two major kernels: a GEMM operation masked by a sparse weight matrix (Masked General Matrix - Matrix multiplication (M-GEMM)), and a dense matrix - sparse matrix multiplication (Dense Matrix - Sparse Matrix Multiplication (DMSpM)).

The mapping on Transmuter is shown in Fig. 3.16. M-GEMM uses a variation of blocked-GEMM, wherein only rows/columns of the dense matrices that generate an element with indices corresponding to NZEs in the weight matrix are fetched and multiplied. DMSpM uses a simplified outer product algorithm similar to SpMM (Sec. 3.8.3) that splits the kernel into DMSpM-Multiply and DMSpM-Merge.

We show the analysis of Sinkhorn on different Transmuter sizes in Fig. 3.17 in order to illustrate the scalability and benefits of reconfiguration. As observed, M-GEMM and DMSpM-Multiply exhibit the best performance in Trans-SC configuration, due to good data reuse across GPEs. In contrast, DMSpM-Merge has optimal performance on Trans-PS, exhibiting a 84.9-98.3% speedup (not shown in the figure) over Trans-SC. Therefore, the

---

**Algorithm 3.9.1** Sinkhorn Distance (MATLAB syntax)

---

```

function SINKHORN(query, data, M,  $\gamma$ ,  $\epsilon$ )
     $\triangleright$  M: distance matrix,  $\gamma$ : regularization parameter,  $\epsilon$ : tolerance

    o = size(M,2);
    H = ones(length(query), o)/length(query);
    K = exp(-M/ $\gamma$ );  $\tilde{K}$  = diag(1./query)K;
    err =  $\infty$ ; U = 1./H;
    while err >  $\epsilon$  do
        V = data./(K'U);
        U = 1./(KV);
        err = sum((U - Uprev)2)/sum((U)2);
    end while
    D = U.*((K.*M)V);
return sum(D)
end function
     $\triangleright$  Sinkhorn Distance between query and data

```

---

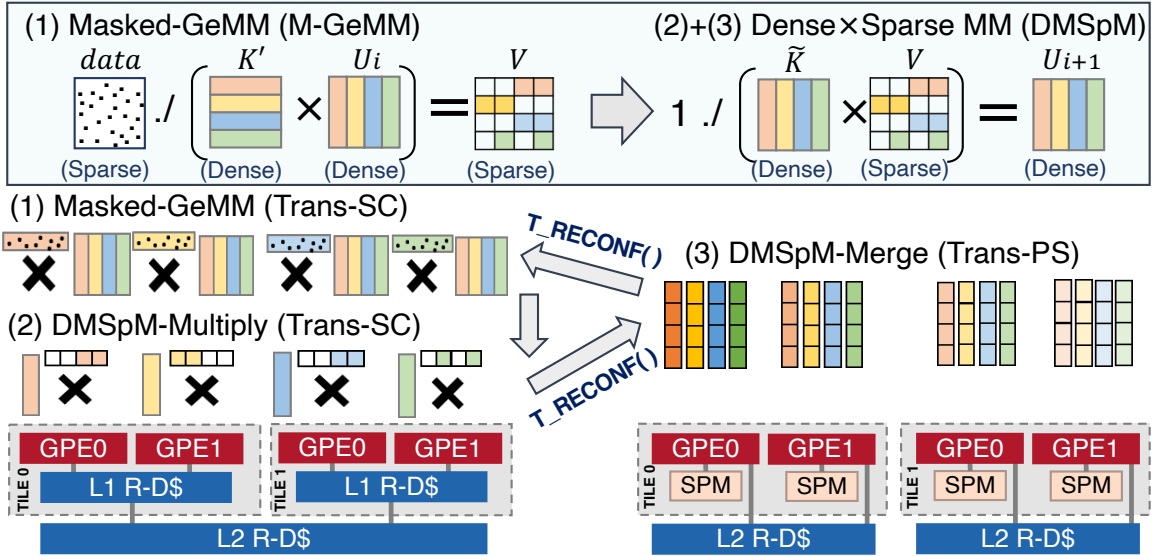


Figure 3.16: Mapping of a multi-kernel, mixed data application, Sinkhorn, on Transmuter. Computation iterates between M-GEMM and DMSpM, with Trans-SC  $\leftrightarrow$  Trans-PS re-configuration before and after DMSpM-Merge. DMSpM-Merge benefits from the private SPMs in Trans-PS, since each GPE works on multiple disjoint lists.

optimal Sinkhorn mapping involves *two reconfigurations* per iteration: Trans-SC  $\rightarrow$  Trans-PS before the start of DMSpM-Merge, and Trans-PS  $\rightarrow$  Trans-SC at the end of it, for the next M-GEMM iteration. Recall from Sec. 3.5 that the reconfiguration time is  $\sim 10$  cycles, and hence does not perceptibly impact the performance or energy. Cache flushing (net 0.2% of the total execution time) is required for M-GEMM but not DMSpM, as DMSpM uses a streaming algorithm. Overall, dynamic reconfiguration results in 47.2% and 96.1% better performance and Energy-Delay Product (EDP), respectively, over Trans-SC-only for the  $4 \times 16$  Transmuter. A heterogeneous solution is also compared against, where M-GEMM



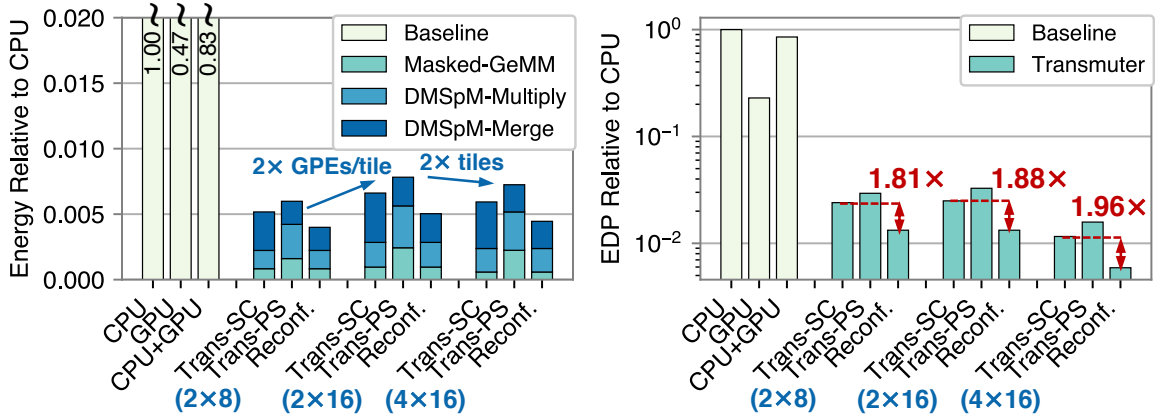


Figure 3.17: Per inner-loop iteration energy (left) and EDP (right) comparing Trans-SC, Trans-PS and Reconf. (Trans-SC  $\leftrightarrow$  Trans-PS) for Sinkhorn normalized to CPU. Input matrix dimensions and densities are — *query*:  $(8k \times 1)$ , 1%, *data*:  $(8k \times 1k)$ , 1%, *M*:  $(8k \times 8k)$ , 99%.

is done on the CPU and DMSpM on the GPU, but this implementation is bottlenecked by CPU  $\rightarrow$  GPU data transfers. As derived from Figure 3.17, the  $4 \times 16$  Transmuter achieves  $38.8 \times$  and  $144.4 \times$  lower EDP than the GPU and heterogeneous solutions, respectively.

### 3.9.5 Throughput and Bandwidth Analysis

We investigate here the impact of scaling the number of tiles ( $N_T$ ) and GPEs per tile ( $N_G$ ) for an  $N_T \times N_G$  Transmuter. Fig. 3.18 illustrates the scaling of Transmuter running the largest input sizes for GEMM, GEMV, and SpMM. GEMM shows near-linear performance scaling with the GPE count. The bandwidth utilization, however, does not follow the same trend as it is dependent on the data access pattern at the shared L2 R-DCache that influences the L2 hit rate. GEMV exhibits increased bank conflicts in the L1 shared cache upon scaling up  $N_G$ , e.g. from  $32 \times 32$  to  $32 \times 64$ . Thus, the performance scaling shows diminishing returns with increasing  $N_G$ , but scales well with increasing  $N_T$ . The performance of SpMM scales well until the bandwidth utilization is close to peak, at which point bank conflicts at the HBM controllers restrict further gains. SpMV follows the trend of GEMV, while FFT and convolution, show near-linear scaling with increasing system size (not shown in Fig. 3.18). In summary, both the bandwidth and throughput exhibit good scaling with Transmuter system size. We use this data to construct analytical performance models for each kernel, which we adopt for fair comparisons with the CPU and GPU.

We also discuss some takeaways from our cache bandwidth analysis for the best-performing Transmuter configuration. GEMM exhibits a high L1 utilization (20.4%) but low L2 utilization (2.7%), as most of the accesses are filtered by the L1. In contrast, SpMM and

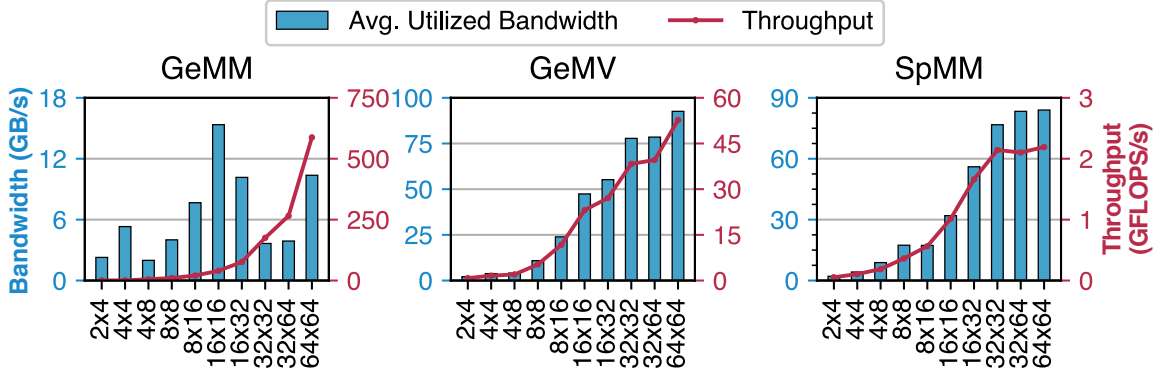


Figure 3.18: Effect of scaling tiles and GPEs per tile on performance and memory bandwidth for GEMM (Trans-SC), GEMV (Trans-SC) and SpMM (Trans-PS). Inputs are: 1k (GEMM), 8k (GEMV), 4096, 0.64% (SpMM).

SpMV in Trans-PS and Trans-SA modes, respectively, have higher L2 utilizations of 68.5-90.5%. The linear algebra kernels show a relatively balanced utilization across the banks, with the coefficient of variation ranging from 0-10.1%. In contrast, both FFT and convolution have a skewed utilization, due to the layout of twiddle coefficients in the SPM banks for FFT, and the small filter size for convolution.

### 3.9.6 Design Space Exploration

We performed a design space exploration with the mapped kernels to select R-DCache sizes for Transmuter. Sizes of 4 kB per bank for both L1 and L2 show the best energy efficiency for all kernels except SpMV. SpMV in Trans-SA benefits from a larger L2 private cache that lowers the number of evictions from fetching discrete packed matrix rows (recall that in Trans-SA, all GPEs in a tile access the same L2 bank). Other kernels achieve slim speedups with larger cache capacities. The dense kernels already exhibit good hit rates due to blocking and prefetching in Trans-SC. SpMM is bottlenecked by cold misses due to low reuse. FFT has a 3.0 $\times$  speedup with 64 kB L1/L2, compared to 4 kB L1/L2, as the number of coefficients stored on-chip scales with L1 size. But, this is outweighed by a 6.4 $\times$  increase in power. Other parameters such as work and status queue depths were chosen to be sufficiently large such that the GPEs are never idled waiting on the LCP.

### 3.9.7 Control Divergence and Data Reuse Analysis

In Section 3.2.2, we characterized some fundamental kernels based on their *control divergence*, *data reuse*, and *arithmetic intensity*. We now build an intuition around the architectural advantages of Transmuter over a GPU for applications with notable contrast in these

characteristics. We implement a parallel microbenchmark on Transmuter and the GPU that allows independent tuning of the divergence and reuse. Fig. 3.19 (left) illustrates this application and the caption details the mechanism. The reuse ( $R$ ) is controlled by the size of the coefficient array, while the divergence ( $D$ ) scales with the number of bins, since threads processing each input element apply functions unique to a bin.

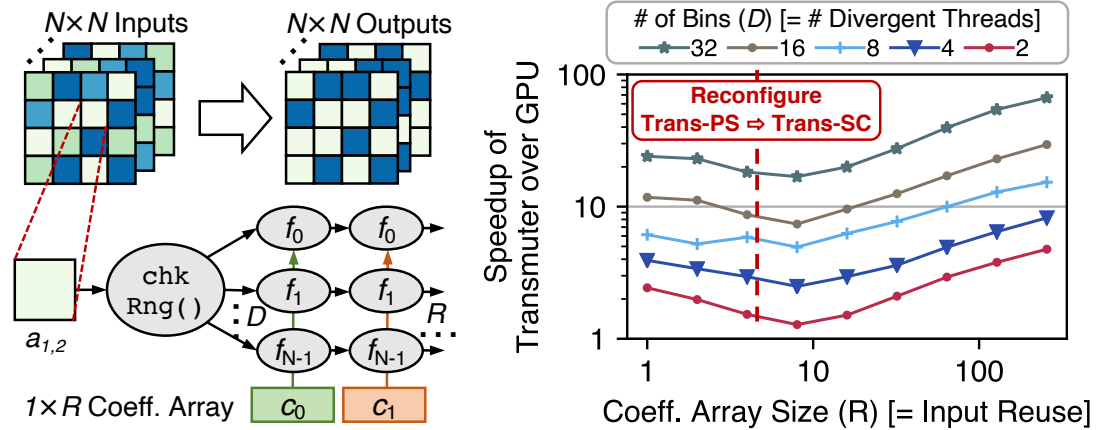


Figure 3.19: *Left*: A synthetic parallel application that launches threads to process  $N \times N$  matrices. Each thread ( $i$ ) reads the input value and bins it into one of  $D$  bins, (*ii*) applies  $R$  instances of function  $f_d$  unique to bin  $d$  and writes the result. Each element of a *coefficient* array feeds into  $f_d$ . Thus the input is reused  $R$  times and the degree of divergence scales with  $D$ . *Right*: Speedup of Transmuter with a uniform-random matrix (# GPEs = # GPU threads = 64). Transmuter reconfigures from Trans-PS to Trans-SC beyond  $R = 4$ .

While this is a synthetic application, it is representative of real-world algorithms that perform image compression using quantization. We execute this microbenchmark with a batch of 1,000  $32 \times 32$  images on a  $4 \times 16$  Transmuter design and compare it with the GPU running 64 threads (2 warps, inputs in shared memory) to ensure fairness. Fig. 3.19 (right) presents two key observations:

- The speedup of Transmuter roughly doubles as the number of divergent paths doubles. This is because threads executing different basic blocks get serialized in the SIMT model (as they are in the same warp), whereas they can execute parallel in SPMD.
- Transmuter has the inherent flexibility to reconfigure based on the input size. In this example, Trans-PS is the best-performing until  $R = 4$ . Beyond that, switching to Trans-SC enables better performance – up to  $7.4 \times$  over Trans-PS – as the benefit of sharing the coefficient array elements across the GPEs in Trans-SC outweighs its higher cache access latency.

**Takeaways.** The SPMD paradigm in Transmuter naturally lends itself well to kernels exhibiting large control divergence, and its ability to reconfigure dynamically allows it to perform well for very low- and high-reuse, and by extension mixed-reuse, workloads.

### 3.10 Related Work

A plethora of prior work has gone into building programmable and reconfigurable systems in attempts to bridge the gap between flexibility and efficiency in architecture design. A qualitative comparison of our work over related designs is shown in Table 3.8. Transmuter differentiates by supporting two different dataflows, reconfiguring faster at a coarser granularity, and supporting a COTS ISA/compiler.

**Reconfigurability.** A few prior works reconfigure at the sub-core level [135, 84, 101, 36, 162] and the network level [72, 105, 185, 147]. In contrast, Transmuter uses native in-order cores and the reconfigurability lies in the memory and interconnect. Some recent work propose reconfiguration at a coarser granularity [126, 6, 162, 36]. Composite Cores [133] proposes a big.LITTLE architecture and techniques to switch execution threads between the cores. PipeRench [70] builds an efficient reconfigurable fabric and uses a custom compiler to map a large logic configuration on a small piece of hardware. HRL [64] is an architecture for near-data processing, which combines coarse- and fine-grained reconfigurable blocks into a compute fabric. The Raw microprocessor [185] implements a tiled architecture focusing on developing an efficient, distributed interconnect. Stream Dataflow [147] and SPU [36] reconfigure at runtime, albeit with non-trivial overheads to initialize the Data-Flow Graph (DFG) configuration. Transmuter, on the other hand, relies on flexible memories and interconnect that enable fast on-the-fly reconfiguration, thus catering to the nature of the application.

**Flexibility.** Prior work has also delved into efficient execution across a wide range of applications. Plasticine [162] is a reconfigurable accelerator for parallel patterns, consisting of a network of Pattern Compute/Memory Units (custom Single Instruction, Multiple Data (SIMD) FUs/single-level SPM) that can be reconfigured at compile-time. Stream Dataflow [147] is a new computing model that efficiently executes algorithms expressible as DFGs, with inputs/outputs specified as streams. The design comprises a control core with stream scheduler and engines, interfaced around a custom, pipelined FU-based CGRA. SPU [36] targets data-dependence using a stream dataflow model on a reconfigurable fabric composed of decomposable switches and PEs that split networks into finer sub-networks. The flexibility of Transmuter stems from the use of general-purpose cores and the reconfigurable memory subsystem that morphs the dataflow and on-chip memory,

thus catering to both inter- and intra-workload diversity.

Table 3.8: Qualitative comparison with prior work.

Architecture	PE Compute Paradigm	Dataflow	Compiler Support	Reconfiguration Granularity	On-chip Memory
Plasticine [162]	SIMD	Spatial	DSL	Pipeline-level, compile-time	SPM
Stream Dataflow [147]	SIMD	Stream	ISA extension	Network-level, run-time	SPM+FIFO
SPU [36]	SIMD	Stream	ISA extension	Network-/Sub-PE-level, run-time	Compute-enabled SPM+FIFO
Ambric [75]	MIMD/SPMD	Demand-driven	Custom	Network-level, run-time	SPM+FIFO
RAW [185]	MIMD/SPMD	Demand-driven	Modified COTS	Network-level, run-time	Cache
<b>Transmuter [this work]</b>	<b>MIMD/SPMD</b>	<b>Demand-driven/Spatial</b>	<b>COTS</b>	<b>Network-/On-chip-memory-level, run-time</b>	<b>Reconfig. Cache/SPM/SPM+FIFO</b>

**Programmability.** There have been proposals for programmable CGRAs that abstract the low-level hardware. Some work develops custom programming models, such as Rigel [99] and MaPU [193]. Others extend an existing ISA to support their architecture, such as Stitch [182] and LACore [177]. Plasticine [162] uses a custom DSL called Spatial [107]. Ambric [75] is a commercial system composed of asynchronous cores with a software stack that automatically maps Java code onto the processor array. Transmuter distinguishes itself by using a standard ISA supported by a simple library of high-level language intrinsics and a COTS compiler, alleviating the need for ISA extensions or a DSL.

### 3.11 Conclusion

This work tackled the important challenge of bridging the flexibility-efficiency gap with Transmuter. Transmuter consists of simple processors connected to a network of reconfigurable caches and crossbars. This fabric supports fast reconfiguration of the memory type, resource sharing, and dataflow, thus tailoring Transmuter to the nature of the workload. We also presented a software stack comprised of drop-in replacements for standard Python libraries. We demonstrated Transmuter’s performance and efficiency on a suite of fundamental kernels, as well as mixed data-based multi-kernel applications. Our evaluation showed average energy-efficiency improvements of  $46.8\times$  ( $9.8\times$ ) over the CPU (GPU) for memory-bound kernels and  $7.2\times$  ( $1.6\times$ ) for compute-bound kernels. In comparison

to state-of-the-art ASICs that implement the same kernels, Transmuter achieves average energy efficiencies within  $9.3\times$ .

## CHAPTER 4

# Intelligent Software and Hardware Reconfiguration for Graph Processing

A key observation obtained when evaluating Transmuter is that the algorithm and the underlying hardware configuration play a crucial role in the performance and efficiency of a kernel implementation on a reconfigurable architecture (Sec. 3.8.4). Based on the observation, we made a bold assumption that the diverse nature of graph analytics makes a great candidate to benefit from a programmable and reconfigurable architecture. Real-world graphs have distinct sizes, densities, and edge distributions. The active vertex set also varies from iteration to iteration for iterative graph algorithms. It has long been a challenge to arrive at a “one-size-fits-all” design for efficient graph processing.

Recent studies have shown that large-scale iterative graph analytics can achieve promising performance on an optimized SpMV backend. This chapter presents CoSPARSE, a software and hardware reconfigurable SpMV framework which makes the best use of Transmuter to accelerate diverse graph algorithms. CoSPARSE explores reconfiguration opportunities in both the software algorithm and the hardware configuration in SpMV implementations on Transmuter. A lightweight partitioning strategy is co-developed to reduce workload imbalance. Heuristic-driven reconfiguration decisions are automatically made by CoSPARSE based on the input data properties and the graph algorithm upon invocation to an SpMV execution. To facilitate programmability, CoSPARSE embeds the SpMV scheduling and implementation in the framework so that end users only need to define key computations to realize a graph algorithm. The combined software and hardware reconfiguration is able to achieve a speedup of up to 2.0x across all benchmarks. Compared to a Xeon CPU executing a state-of-the-art software reconfigurable graph processing framework, CoSPARSE shows up to  $877\times$  better energy efficiency and  $3.51\times$  speedup for a variety of common graph algorithms.

The work presented in this chapter was published as a paper at DAC’21 [54].

## 4.1 Introduction

SpMV is an essential linear algebraic operation that has been widely adopted in many irregular workloads, such as ML and data mining [151]. Recent studies have shown that large-scale iterative graph analytics can achieve promising performance on a high-performance backend optimized for SpMV [181]. However, guaranteeing high performance consistently across different input graphs, graph algorithms, or algorithm iterations, is challenging. First, real-world graphs have distinct sizes and distributions. The adjacency matrices used to represent graphs have sizes scaling from hundreds to billions and densities ranging from  $10^{-7}$  to  $10^{-1}$  [116], leading to dramatically different memory footprints. Second, the active vertex set, *i.e.* the frontier vector, varies from iteration to iteration, causing highly optimized solutions for certain use cases to encounter significant performance loss for the other cases. Therefore, it is hard to arrive at a “one-size-fits-all” design for the efficient execution of graph algorithms [30].

To adapt to different scenarios, prior work has followed two distinct routes: (i) software-level optimizations, *e.g.* deciding a suitable sparse storage format based on the density and size of the input matrix and vector and selecting either a dense or sparse dataflow [172, 217, 30, 212, 196, 194, 206], and (ii) hardware-level optimizations that target specifically SpMV or graph algorithms by focusing on the efficient use of on-chip memory [151, 165, 76, 215]. Merely relying on software optimizations could fail to fully explore on-chip data reuse due to limitations in hardware. On the other hand, hardware-only optimizations are also likely to achieve suboptimal performance for certain graph algorithms and inputs. For example, a hardware accelerator optimized for graph algorithms based on sparse matrix dense vector computations will consume unnecessary compute cycles for those involving sparse vector computations. The ideal design for SpMV-based graph analytics should run the desired algorithm on hardware that is most efficient for the data access pattern based on the input characteristics. This complex and high-dimensional design space, therefore, calls for a reconfigurable SpMV framework that provides both the flexibility to adapt to different inputs and algorithms and a faithful strategy to speedily traverse the available reconfiguration points to achieve the highest achievable performance.

Our proposed solution, CoSPARSE, explores reconfiguration opportunities in both software and hardware, as shown in Figure 4.1. In software, it considers two SpMV algorithms based on inner and outer product. The choice of the algorithm directly affects the access pattern of the input/output data, and the load-balancing strategy. In hardware, reconfigurability is manifested in the on-chip memory hierarchy of the underlying hardware, since SpMV is known to be memory-intensive and is bottlenecked by irregular memory accesses.



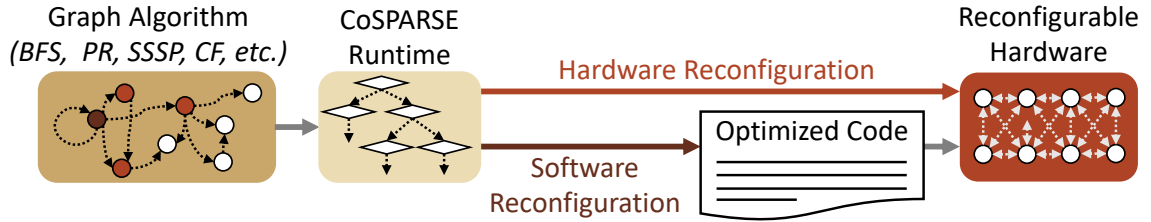


Figure 4.1: Overview of the proposed CoSPARSE framework.

CoSPARSE uses a hardware substrate that supports reconfigurations in both the on-chip memory sharing pattern (shared/private) and on-chip memory type (cache/scratchpad). The software and hardware reconfiguration decisions are made in an integrated dynamic framework, guided by knowledge from extensive experiments and in-depth analysis. On top of software and hardware reconfiguration, CoSPARSE provides a workload-balancing strategy to harness maximum parallelism for irregular sparse matrices. All of these synergistic benefits are showcased on a suite of common iterative graph analytics algorithms, including Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), PageRank (PR), and Collaborative Filtering (CF), constructed on top of CoSPARSE’s SpMV abstraction. Specifically, we make the following contributions:

- On-the-fly, automatic, coordinated reconfiguration of the hardware and software based on the input data properties, *i.e.* the dimensions and densities of matrices and the density of vectors, including:
  - *Software reconfiguration* between inner product and outer product based SpMV implementations, and
  - *Hardware reconfiguration* of the memory subsystem to exploit data-sharing patterns (private/shared) and on-chip memory types (cache/scratchpad).
- Extensive experiments with in-depth analysis to derive the threshold for software and hardware reconfiguration decisions.
- A consistently efficient, high-performance SpMV framework for graph analytics across diverse algorithms and datasets.
- Evaluation of CoSPARSE against competing systems that demonstrates up to  $877\times$  better energy efficiency and  $3.51\times$  speedup for a variety of graph algorithms over Ligra on a Xeon CPU.

## 4.2 Background and Related Work

Graph algorithms can be implemented as iterative SpMVs to take advantage of highly optimized SpMV backends [181]. However, the diverse nature of graph processing workloads creates challenges for achieving high performance across a wide range of graph algorithms and input datasets.

### 4.2.1 Graph Frameworks using Software Reconfigurations

For graph traversal algorithms such as BFS and SSSP, the size of the active vertex set varies from iteration to iteration [206]. For example, the SSSP algorithm on soc-pokec, a commonly used graph benchmark, shows that during execution the percentage of active vertices increases from  $<0.1\%$  to  $47\%$  and again decreases to  $<0.1\%$  (Figure 4.9). To harness this property, switching between dense and sparse representations of the active vertex set and the corresponding dataflows across iterations is widely adopted in recent graph frameworks [172, 217, 30, 212, 196, 194, 206]. In terms of SpMV, the dense representation is equivalent to the inner product algorithm and the sparse one corresponds to the outer product algorithm. Graph frameworks usually target existing platforms with no hardware modifications and require user input for accurate reconfiguration. For example, Ligra [172], a lightweight shared memory based graph framework implementing software reconfiguration, uses an empirical parameter, *i.e.*  $|V| = |E|/20$ , as the reconfiguration threshold unless users set it differently, where  $|V|$  denotes the active vertex size and  $|E|$  is the number of edges. CoSPARSE, instead, automatically analyzes choices at both software and hardware levels within a tightly-coupled framework to achieve the best performance at graph iteration granularity.

### 4.2.2 Optimized Hardware Acceleration for Graph Analytics

Many vertex-programming based graph processing accelerators using frontier scheduling have been proposed recently [73]. Graphicionado [76] exploits the on-chip scratchpad memory for random accesses and applies graph slicing to maximize data reuse. TuNao [215] maps the Gather-Apply-Scatter paradigm to ECGRA modules and enhances data reuse by storing high-degree vertices in on-chip buffers. GraphPIM [144] provides efficient processing-in-memory offloading with minor architectural extensions to achieve dramatic memory bandwidth improvement. To obtain the best efficiency with minimum hardware, graph processing accelerators tend to target one dataflow, and often do not consider the characteristics of the input vector. CoSPARSE, instead, is implemented on top of

a programmable general-purpose hardware substrate that can be easily extended to support different graph algorithms by providing an SpMV framework abstraction and efficiently executes both inner product and outer product.

### 4.2.3 Opportunities in Combining Software/Hardware Optimizations

CoSPARSE requires a hardware substrate that is programmable and reconfigurable to orchestrate software and hardware reconfiguration. Recent work has proposed a many-core general-purpose accelerator called Transmuter [152] that supports reconfiguration of the resource sharing pattern (private/shared), and on-chip memory type (cache/SPM). Transmuter has been demonstrated to accelerate applications in sparse and dense linear algebra [173, 152], signal processing [152], and deep neural networks [203]. The architecture features a massive number of lightweight PEs and a reconfigurable memory hierarchy. The PEs are grouped into tiles and are coordinated by a LCP. Each PE and LCP are lightweight in-order processors with standard ISA support. The PEs are connected to a two-level memory hierarchy consisting of reconfigurable crossbars (R-XBars) and reconfigurable data caches (R-DCaches). Each level of the reconfigurable memory hierarchy (L1/L2) can be configured into shared/private caches/SPMs. The reconfiguration can happen both at compile time or at runtime. For runtime hardware reconfiguration, one of the LCPs would call the reconfiguration API to reconfigure the R-DCaches and R-XBars. The runtime hardware reconfiguration overhead is estimated to be  $\leq 10$  clock cycles. From this point of the chapter, We will refer to an  $A \times B$  system as a Transmuter design with  $A$  tiles and  $B$  PEs per tile. The use of programmable cores facilitates dataflow reconfiguration and support for diverse graph algorithms. The hardware reconfigurability of Transmuter also lends a good fit to CoSPARSE, since the hardware is amenable to different data access patterns and flexible in response to properties of the evolving data set. Note that though CoSPARSE is evaluated on the Transmuter hardware platform, the concept and strategy of this work are applicable to any general programmable reconfigurable hardware.

## 4.3 CoSPARSE Reconfiguration Layer Design

Figure 4.2 gives an overview of the heuristic-driven reconfiguration strategy, which is triggered before each SpMV execution. Based on the density of the input vector, we decide whether to use the inner product or outer product based SpMV algorithm; this is the software (re)configuration choice. Then, based on the density and size of the matrix and the vector, we decide on the two-level on-chip memory configuration of the hardware; this is

the hardware (re)configuration. In the following subsections, we highlight the supported on-chip memory modes and address the load balancing for irregular matrices. We also present the rationales behind the decision-making heuristic by studying the trade-offs in the inner product vs. outer product mapping and the choice of memory/sharing mode. Note that we determine the various thresholds used at each level of our decision tree based on empirical data from our analysis in Sec. 4.3.3.

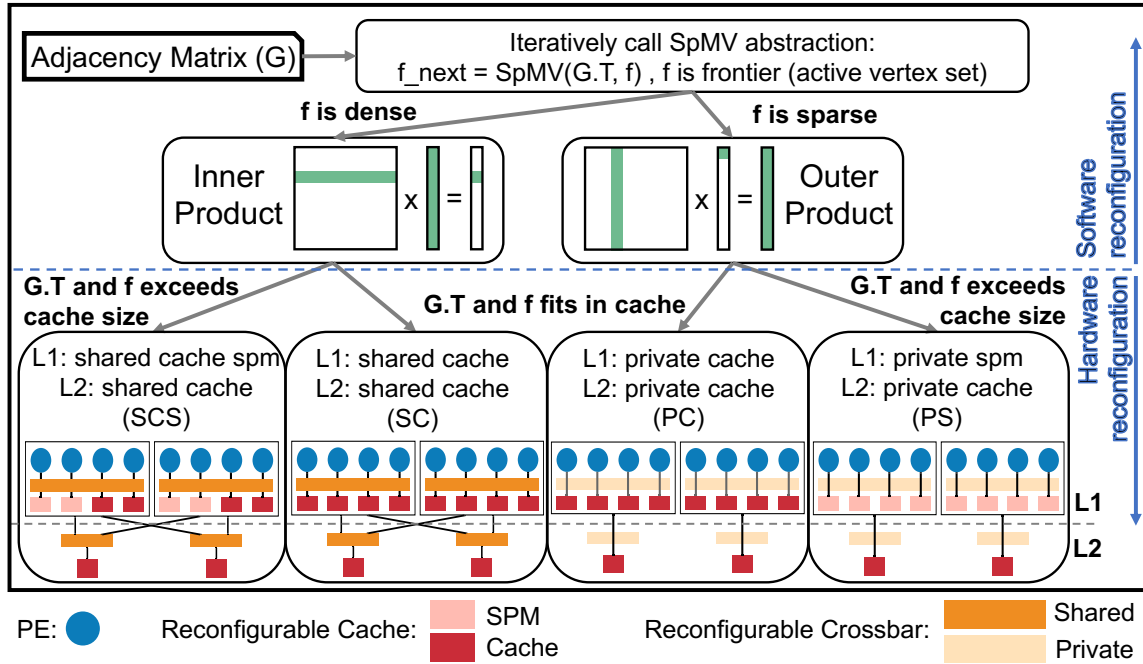


Figure 4.2: Structure of CoSPARSE hardware and software reconfiguration framework. For every invocation to CoSPARSE, we select the best software (inner product or outer product), followed by hardware configurations (Trans-SCS or Trans-SC for inner product, Trans-PC or Trans-PS for outer product), assuming a  $2 \times 4$  system.

### 4.3.1 Reconfigurable SpMV Implementation

Figure 4.2 shows the four hardware configurations that we identified to be most suitable for SpMV, *i.e.* Trans-SC (L1: shared cache and L2: shared cache) and Trans-SCS (L1: shared cache scratchpad and L2: shared cache) for inner product and Trans-PC (L1: private cache and L2: private cache) and Trans-PS (L1: private scratchpad and L2: private cache) for outer product.

**Inner Product Implementation.** To maximize parallelism, the matrix is partitioned into disparate row partitions which are stored in a row-major Coordinate Format (COO) format to facilitate spatial locality for memory accesses. The COO format stores the row

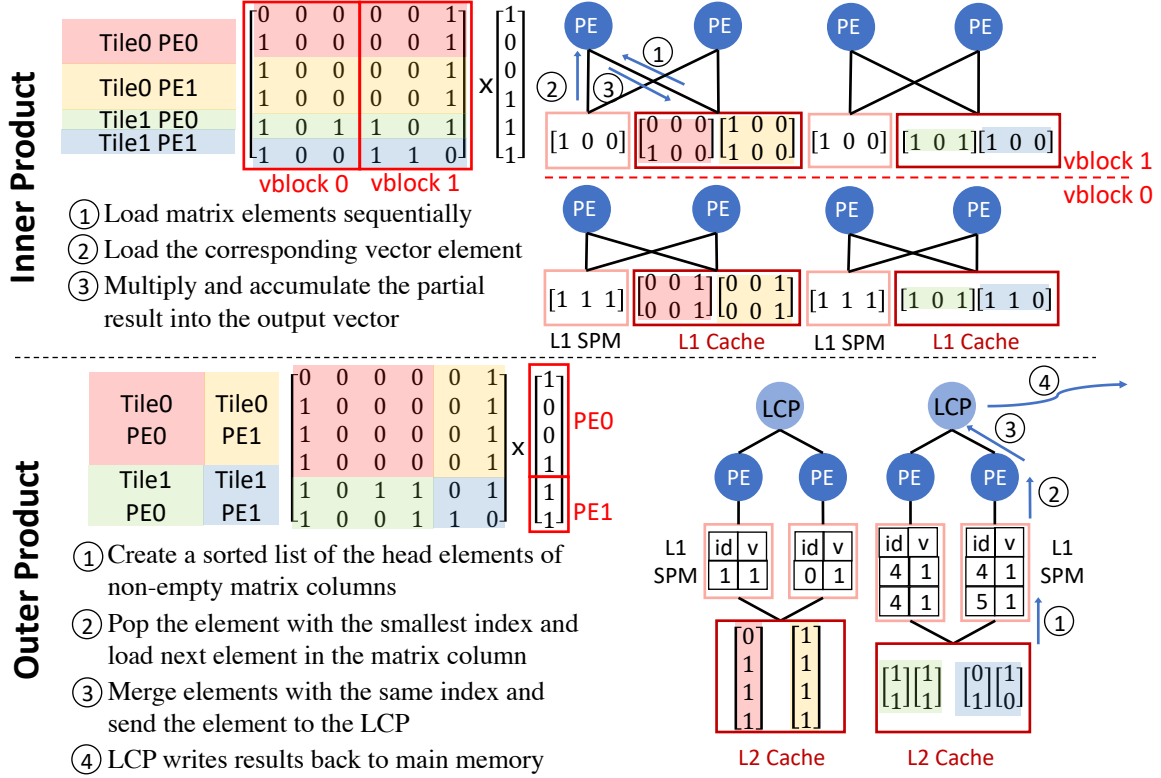


Figure 4.3: Matrix partitioning based on NZEs and algorithm mapping of inner product on Trans-SCS and outer product on Trans-PS that focuses on maximizing data reuse and reducing stalls for random accesses on a  $2 \times 2$  system.

index, column index, and the value for each matrix NZE. The vector is stored as a dense array. Each tile performs multiplication and accumulation on one of the matrix row partitions with the vector. Hence, each tile works on different segments of the output vector in parallel without introducing data races, and thus avoids synchronization overheads. In addition, the mapping exploits reuse opportunities of the input vector, which is shared among the tiles and PEs within a tile. Therefore, to maximize data sharing, CoSPARSE selects the Trans-SC and Trans-SCS modes, which enables the PEs and the tiles to share a large chunk of on-chip memory.

Figure 4.3 (top) illustrates the computation scheme of inner product for Trans-SCS. The input vector elements are stored in the shared SPM in L1 to curtail the overhead of random accesses to the vector elements due to matrix sparsity. The vector elements in the SPM are shared among all PEs within a tile. For large matrices, the matrix is partitioned vertically to ensure that the vector segment corresponding to a vertical partition (vblock) fits in the SPM. Trans-SC uses the same scheduling except that the vector elements are randomly accessed from the L1 shared caches.

**Outer Product Implementation.** Similar to inner product, outer product also involves each tile multiplying an exclusive row partition with the vector. The matrix is stored in a column-based sparse format, *i.e.* Compressed Sparse Column Format (CSC) format, which stores the row index and the value for each matrix NZE and an array of pointers to the start row index of each column. The vector is stored in a sparse format, *i.e.* (*index, value*) tuples of the vector NZEs. The LCP assigns a contiguous chunk of vector NZEs to each PE and the PEs perform mergesort with the corresponding matrix columns. Since each PE accesses an exclusive set of columns, there is no data sharing between PEs and between tiles. Therefore, private on-chip memories are used in both L1 and L2 to prevent data thrashing and cache contamination. The PEs can also benefit from higher access bandwidth and shorter latency to L1, since bank conflicts and arbitration in the crossbars are eliminated.

Figure 4.3 (bottom) shows the execution flow of outer product in Trans-PS. The sorted list maintaining the head elements of the non-empty matrix columns is kept in the private SPM to support fast random accesses from list management. For higher scalability, the sorted list uses a heap structure, *i.e.* a binary tree which guarantees that the parent is smaller than its children. When the sorted list cannot fit in the SPM, it spills over to the shared memory, but the tree nature of the heap structure ensures that the majority of comparisons and swaps still happen in the SPM. The scheduling for Trans-PC is the same. However, since Trans-PC uses caches in L1 and has no control over the cache replacement policies, the sorted list elements may be evicted to L2 or even the main memory.

### 4.3.2 Workload Balancing Strategies

Many real-world sparse matrices have non-uniform distributions [30], causing imbalanced workload distribution across PEs. To achieve maximum parallelism through workload balancing, both static matrix partitioning (before execution) and dynamic task distribution (during execution) are applied.

**Inner product** treats the vector as dense, so the execution time of a PE is highly dependent on the number of matrix NZEs assigned to it. Figure 4.3 (top) illustrates the matrix partitioning method used by inner product. The sparse matrix is first statically partitioned into row partitions with the same Number of Non-Zeros (NNZ). Each PE is assigned one of the row partitions and thus obtains a similar amount of work. The row partitions are further divided into multiple vertical blocks (vblocks) so that the vector elements corresponding to each vblock can fit in the shared SPM. Ideally, PEs work on the same vblock at a time so that each tile can fetch the vector elements for the other tiles into L2 caches. The vertical partition is not required for the Trans-SC mode but can still be beneficial because of the im-

proved spatial and temporal locality of vector accesses. Since each tile works on disparate row partitions, no synchronization is needed after each PE finishes processing a vblock. Also, since the matrix is sparse, the imbalance in the NNZ *within* a vblock is not large enough to visibly impact performance. The proposed partitioning scheme is sufficiently lightweight and effective in that it balances the workload by assigning each PE the same number of matrix NZEs and fully utilizes the underlying hardware by considering the size of the on-chip storage.

**Outer product** is different from inner product in that the vector density affects the workload (actual NNZ) assigned to PEs. If the vector used for SpMV remains the same throughout execution, the matrix partitioning can also take into account vector sparsity. However, this is not the case for our target applications, *i.e.* iterative graph algorithms, and thus dynamic workload-balancing is needed. Similar to inner product, the matrix is first divided into row partitions with the same number of NZEs and assigned to each tile. Within a tile, the LCP distributes the NZEs of the vector evenly to each PE, such that the number of columns assigned to each PE, *i.e.* the storage needed for the sorted list, is roughly the same. The combination of static and dynamic workload balancing provides an effective solution for irregular matrix distribution and works well for applications with evolving vectors, *e.g.* graph algorithms.

### 4.3.3 Reconfiguration Threshold Analysis

The thresholds used at each level of the reconfiguration decision tree, *i.e.* software and hardware reconfiguration, are based on extensive experiments and analysis. The methodology for these experiments is detailed later in Sec. 4.4.

#### 4.3.3.1 Software Reconfiguration Threshold

When the vector is sparse, outer product tends to outperform inner product because it only considers the matrix columns that have corresponding NZEs in the input vector, and thus significantly reduces the number of matrix elements fetched during computation. However, the overhead of mergesort grows in a super-linear fashion with the number of matrix columns to merge, which grows with increasing vector density and matrix dimension, and causes the benefits of outer product to diminish in comparison to inner product.

Figure 4.4 shows the speedup of outer product over inner product and demonstrates a clear crossover point between the two algorithms for different system sizes and input matrices. We define the *crossover vector density (CVD)* as the density above which the inner product algorithm should be used, and below which, the outer product algorithm

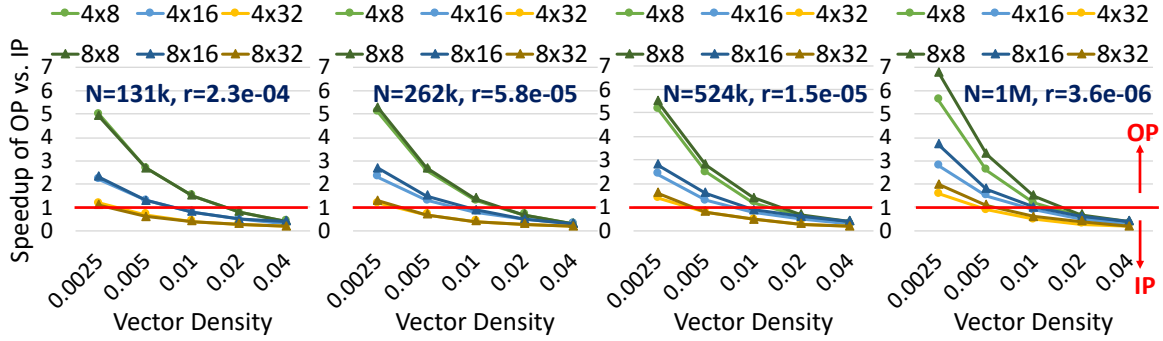


Figure 4.4: Speedup of outer product (Trans-PC) vs. inner product (Trans-SC). Generally, inner product performs better for dense vectors and outer product performs better for sparse vectors. The crossover vector density decreases when more PEs are present in a tile.

should be used to achieve the best performance. The CVD decreases with an increasing number of PEs per tile because the performance of outer product does not scale with the number of PEs as well as it does for inner product. Since the LCP does the final round of merging and serves as the serialization point, increasing the number of PEs improves the parallelism of the first round of merge sort but the performance of the outer product is then bottlenecked by the serial processing performance of the LCP.

The dimension and density of the matrix also have an impact on the CVD. When the matrix becomes sparser, the total amount of reuse for vector elements becomes smaller for inner product, whereas outer product is not affected by the matrix sparsity, causing the CVD and the performance benefit of outer product to increase slightly.

**Takeaways.** *There exists a crossover point at which CoSPARSE switches from inner product to outer product to achieve the best performance as the vector density decreases. The crossover density decreases from  $\sim 2\%$  to  $\sim 0.5\%$  as the number of PEs in a tile increases from 8 to 32.*

#### 4.3.3.2 Hardware Reconfiguration Threshold for Inner Product

The best hardware reconfiguration for inner product depends on both the dimension and density of the matrix, as well as the density of the vector. Although the vector is stored in a dense format, the MAC operation and the write-back of the partial sum are bypassed when the vector element is zero. Therefore, as the vector density increases, the execution times for both the Trans-SC and Trans-SCS modes increase due to the increasing number of floating point computations and store operations. As shown in Figure 4.5, the performance benefit of the Trans-SCS mode is positively correlated to the vector density. In the Trans-SC mode, the vector elements are fetched into L1 caches on-demand and could be evicted to L2



caches or even the main memory by the cache replacement policy. The Trans-SCS mode, on the other hand, stores the vector elements in the L1 SPM to allow fast random accesses. Since Trans-SCS eliminates the case where useful vector elements are evicted from L1 and reloaded afterward, Trans-SCS encounters a lower number of L2 cache accesses than Trans-SC mode and thus fewer cache misses and fewer memory stalls, especially for high-density vectors. The reduced L2 cache accesses also make the execution time on Trans-SCS increase more slowly as the vector density grows and thus increases the performance benefit of Trans-SCS mode.

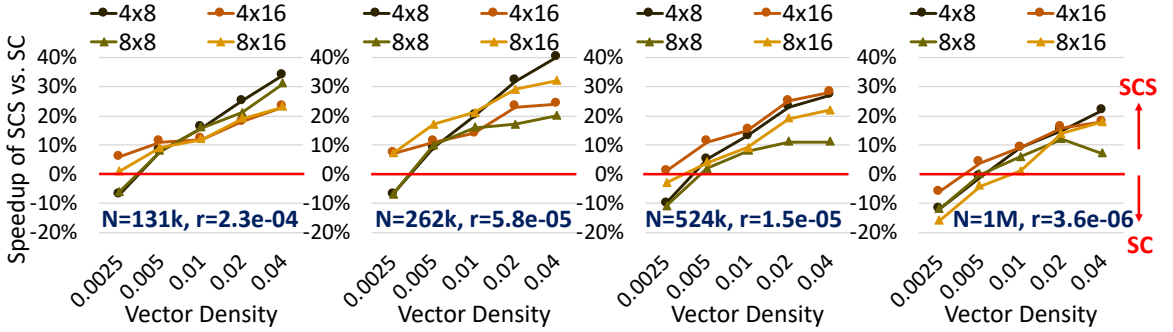


Figure 4.5: Speedup of Trans-SC vs. Trans-SCS for inner product. Trans-SCS achieves more performance gain for denser vectors or when the reuse of data in SPM is higher.

The matrices evaluated here have the same NNZ, so the largest matrix is also the sparsest matrix. The performance benefit obtained by Trans-SCS is highly dependent on the number of times the vector elements in the SPM are reused ( $N_{reuse}$ ). For uniformly random matrices,  $N_{reuse}$  is proportional to the NNZ in a vblock (Figure 4.3), *i.e.*  $\frac{N \cdot r \cdot \text{NUM\_PES\_PERTILE}}{\text{NUM\_TILES}}$ , where  $N$  is the matrix dimension and  $r$  is the matrix density. Based on the formula, the largest matrix exhibits the least reuse among the four matrices, and thus the least speedups. For the same reason, the performance benefit reduces when the system size changes from  $4 \times 8$  to  $8 \times 8$  or from  $4 \times 16$  to  $8 \times 16$ , since  $N_{reuse}$  decreases as the number of tiles increases. When the number of PEs increases,  $N_{reuse}$  also increases. However, as the Trans-SC mode also has a larger cache to fit more vector elements in L1, the performance benefit does not show a clear trend.

**Takeaways.** *The speedup of Trans-SCS is positively correlated to vector density as well as the number of times that the vector elements stored in the SPM are reused, *i.e.* the number of matrix elements corresponding to these vector elements.*

### 4.3.3.3 Hardware Reconfiguration Threshold for Outer Product

The speedup of Trans-PS over Trans-PC is reported in Figure 4.6. A key observation is that the performance benefit of Trans-PS is closely related to the number of columns that need merging, which is determined primarily by the matrix dimension and vector density. Among the matrices evaluated in Figure 4.6, the matrix with a size of 131k has the least dimension, and the Trans-PS mode underperforms compared to Trans-PC when the vector density is less than 0.01. However, Trans-PS always outperforms Trans-PC for the matrix with a size of 1M, the largest and most sparse matrix. As the vector density increases, more matrix columns need to be merged, resulting in an increase in speedup with Trans-PS. This is because Trans-PS maintains the sorted list of the head elements of the non-empty matrix columns in a heap structure in the SPM, and the majority of random accesses are handled by the SPM. Trans-PC, however, does not have control over the locations of the sorted list elements. When the sorted list cannot fit in L1, the list management accesses can span across the memory hierarchy. The situation becomes severe with high vector density since the length of the sorted list grows with the vector density, which is indicated by the lower hit rates of both the L1 and L2 caches. On the other hand, when the vector sparsity allows the sorted list to fit in the L1, Trans-PC outperforms Trans-PS as Trans-PC does not have SPM management overheads.

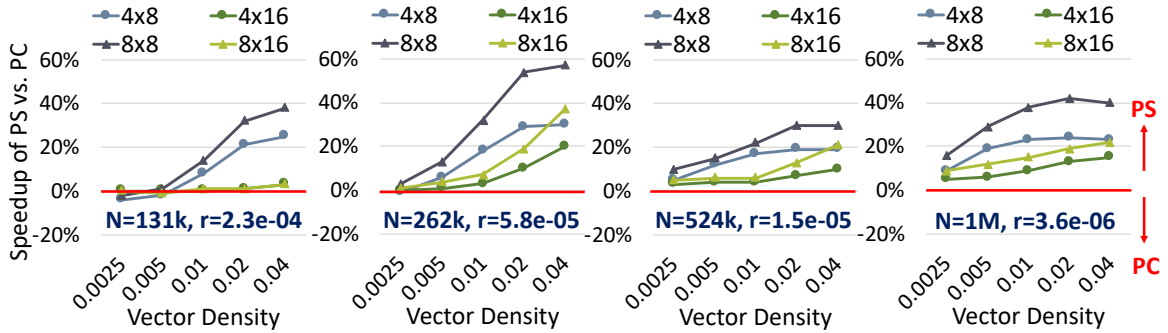


Figure 4.6: Speedup of Trans-PC vs. Trans-PS for outer product. The performance gain of Trans-PS grows with increasing vector density, increasing number of tiles, and decreasing number of PEs per tile.

The best-performing configuration is also related to the size of the hardware system. Since the number of PEs and L1 RCache banks are the same, the increased number of PE in a tile allows Trans-PC to have a larger cache to fit the sorted list. As the L1 hit rates increase for Trans-PC, the speedup of Trans-PS drops when there are more PEs per tile. On the other hand, the performance benefit of Trans-PS increases rapidly with the number of tiles. As the number of cores doubles by switching from a  $4 \times 8$  to an  $8 \times 8$  system, the Trans-PC mode achieves an average speedup of  $1.80 \times$  and the Trans-PS mode achieves

1.96 $\times$ . Increasing the number of tiles keeps the number of matrix columns to merge the same, but reduces the length of the matrix columns, and thus the total NNZ to merge. In this case, the performance benefit of the Trans-PS mode becomes more obvious, because the chances of loading the next elements in the matrix column are reduced, and the random accesses to the sorted list become a more significant bottleneck.

*Takeaways.* *Trans-PS achieves better performance when there are more columns to merge, or when the length of columns to merge reduces. The speedup of Trans-PS decreases for systems with more PEs in a tile.*

### 4.3.4 Graph Analytics Algorithms on CoSPARSE

Hardware-accelerated graph processing solutions often require programmers with in-depth architectural knowledge of the hardware to fully exploit the available performance benefit [76]. Existing graph processing frameworks, on the other hand, enhance user-friendliness by abstracting away scheduling and implementation details, but achieving the best performance still requires expert intervention, *e.g.* to define accurate thresholds [212]. CoSPARSE addresses both performance and programmability with a software and hardware reconfigurable SpMV framework. The software and hardware configurations are automatically determined based on algorithms and input characteristics upon invocation of the decision tree. The runtime hardware reconfigurations are triggered by one of the LCPs and are estimated to take  $\leq 10$  cycles. The SpMV scheduling and implementation are embedded in the framework. End users only need to define the key computations to realize a graph algorithm, similar to [172]. Example algorithm implementations are shown below.

#### 4.3.4.1 Graph Analytics Algorithm Mapping

To map a graph algorithm to CoSPARSE, two key operations need to be specified. **Matrix\_Op** defines the computation between the NZEs of the adjacency sparse matrix and the elements of the frontier vector. **Vector\_Op** applies computation to the vector elements. Taking SpMV as an example, Matrix\_Op denotes the dot product between a matrix row and the vector. Since Matrix\_Op already calculates the final result, Vector\_Op is not applicable for SpMV. All graph algorithm implementations in CoSPARSE are mapped based on code from the Ligra framework [172].

In this work, we implement and evaluate four common graph algorithms which are representative of machine learning and graph traversal, *i.e.* BFS, SSSP, PR, and CF. The definitions of the key operations of these graph algorithms are detailed in Table 4.1.

**BFS** is a graph search algorithm used in social network analysis, GPS navigation, web

Table 4.1: Definitions of Matrix\_Op and Vector\_Op of Algorithms mapped to CoSPARSE, where Sp represents the adjacency sparse matrix and V represents the frontier vector. *src* is the source vertex and *dst* is the destination vertex.

Algorithm	Matrix_Op(Sp,V)	Vector_Op(V)
SpMV	$\sum \text{Sp}_{src,dst} * V_{src}$	N/A
BFS	$\min(V_{src})$	N/A
SSSP	$\min(V_{src} + \text{Sp}_{src,dst}, V_{dst})$	N/A
PR	$\sum (V_{src}/\text{deg}(src))$	$\alpha + (1 - \alpha) * V_{updated\_dst}$
CF	$\sum (\text{Sp}_{src,dst} - V_{src} * V_{dst}) * V_{src} - \lambda * V_{dst}$	$\beta * V_{updated\_dst} + V_{dst}$

page crawling, *etc.* BFS [206] explores all connected vertices of a graph from a start vertex. At each iteration, each unvisited destination is updated by its first active source vertex. Matrix\_Op checks the active sources of target destinations; if active, it updates the values in the vector for the next iteration.

SSSP [172] is a graph traversal algorithm that computes the distance between a single source and all other vertices in a weighted graph. The algorithm is used in many applications such as finding driving directions in maps or computing the min-delay path in telecommunication networks. Matrix\_Op computes the distance between the source to all destinations of an active vertex and uses the minimum to update the frontier vector. In contrast to BFS, the result of SSSP depends on the edge weights.

PR [172] is used to rank web pages based on specific metrics (*e.g.* popularity) by computing the probability that a hyperlink (*i.e.* edge) would end in a particular page (*i.e.* vertex). In PR, all vertices are always active. In each iteration, Matrix\_Op calculates an influence value for each vertex based on the popularity of its incoming neighbors. Vector\_Op adds a constant to the vertex properties updated by Matrix\_Op.

CF [181] is a machine learning algorithm used by recommendation systems, which estimates users' rating for a given item based on an incomplete set of (user, item) ratings. The users' ratings are based on a set of hidden/latent features and each item can be expressed as a combination of these features. The CF implementation [171] here is accomplished using gradient descent, where Matrix\_Op computes the gradient and Vector\_Op applies the gradient to the frontier vector.

#### 4.3.4.2 Input and Output Conversion Overhead

Throughout the execution of a graph analytics algorithm, the sparse matrix remains constant, but the sparsity of the vector may vary from iteration to iteration. A new output vector is produced and serves as the input vector for the next iteration. To support the inner product and outer product algorithms and runtime reconfiguration, two copies of the in-

put compressed sparse matrix (in COO and CSC formats, respectively) are stored in main memory to avoid matrix conversion overhead, similar to [172], whereas the lightweight vector conversion between sparse and dense format is performed for the iterations that require reconfiguration. The overhead of vector conversion, however, is minimal compared with the total execution time of graph algorithms and is therefore performed during execution when needed. In most of the graph analytics algorithms in our experiments, switching between inner product and outer product only happens once or twice during execution, *e.g.* for BFS and SSSP, where the vector changes from sparse to dense and then back to sparse. The other algorithms, namely PR and CF, always use dense vectors, and thus no format conversion is needed.

## 4.4 Methodology

CoSPARSE is modeled using the gem5 simulator [20]. The microarchitectural parameters are listed in Table 3.3. The PEs and LCPs are modeled after an in-order ARM Cortex M4F, and the cache and crossbar latencies are based on prior work [152]. For systems larger than  $8 \times 16$ , the simulation resources required become prohibitive and a trace-based simulation model is used [152].

A power model is built based on the static and dynamic power of each individual component of the system and cross-verified with a fabricated chip prototype [153]. The crossbar and core power models are based on synthesis reports and cache power is calculated by CACTI 7.0 [142].

The SpMV implementation in CoSPARSE is compared against state-of-the-art SpMV implementations on a CPU (Intel i7-6700K) running MKL 2018.3 and a GPU (NVIDIA Tesla V100) running cuSPARSE v8.0. The graph algorithm implementations are evaluated against Ligra [172]. To evaluate the performance and efficiency of CoSPARSE, we use a combination of uniformly random matrices, power-law matrices generated by NetworkX, and real-world graphs from SNAP dataset [124] and SuiteSparse Matrix Collection [40]. The details of the real-world graphs are listed in Table 4.2.

## 4.5 Evaluation

In this section, we first evaluate the benefits of the proposed workload balancing techniques. Then, we compare the performance of standalone SpMV to that of state-of-the-art implementations on the CPU and GPU. To showcase the automatic reconfiguration, we present a case study to illustrate the execution of a graph algorithm on CoSPARSE. Finally,

Table 4.2: Specifications for real-world graphs.

Graphs	# Vertices	# Edges	Type	Kind	Density
livejournal	4,847,571	68,992,772	Social Network	Directed	$2.9 \times 10^{-6}$
pokec	1,632,803	30,622,564		Directed	$1.2 \times 10^{-5}$
youtube	1,134,890	2,987,624		Undirected	$2.3 \times 10^{-6}$
twitter	81,306	1,768,149		Directed	$2.7 \times 10^{-4}$
vsp	21,996	2,442,056	Random	Undirected	$5.0 \times 10^{-3}$

we compare the performance across a set of common iterative graph analytics algorithms with that of Ligra running on a server-class CPU.

### 4.5.1 Workload Balancing Evaluation

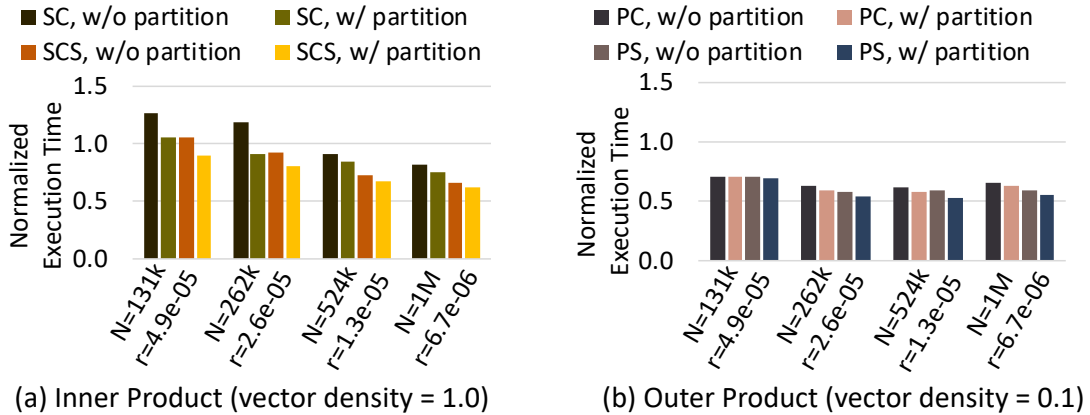


Figure 4.7: The SpMV execution time of power-law matrices normalized to uniform matrices on Trans-SC (inner product) and Trans-PC (outer product) on an  $8 \times 16$  system. Workload balancing benefits inner product more than outer product, especially Trans-SC for inner product.

The execution time of SpMV for power-law matrices, normalized to that for uniformly random matrices of the same dimension and density on cache-only hardware configurations, is shown in Figure 4.7. For inner product, the workload-balancing technique improves the execution time by 7% to 30% and benefits Trans-SC more than Trans-SCS. Since Trans-SC does not have SPMs, it cannot efficiently handle random accesses. In addition, in Trans-SC, the workload imbalance could cause some PEs to finish their assigned work early and remain idle, instead of fetching vector elements that could be reused by other PEs into the shared L1. Therefore, the performance of Trans-SC is more sensitive to the irregular matrix distribution, and thus more likely to benefit from the workload-balancing scheme. It is worth noting that in some cases for inner product, the execution time of power-

law matrices is less than that of uniform matrices. This is because the existence of dense rows/columns in power-law matrices results in fewer non-empty matrix rows/columns. In this case, fewer input vector elements are used for computation and fewer output vector elements are generated, which are more likely to fit in the L1, improving both locality and performance.

As shown in Figure 4.7-b, for outer product, the execution time of power-law matrices is also shorter than that of uniformly random matrices. This is because the irregular distribution of the matrices increases the possibility that the matrix column corresponding to a vector NZE has no elements, thus reducing both the number of columns and the NNZ to merge. The matrix partitioning technique further improves the execution time of both hardware configurations by up to 10%.

## 4.5.2 Comparison against Existing Platforms

The hardware substrate used in CoSPARSE is programmable so as to support easy implementations of SpMV-based applications, such as graph algorithms. Therefore, we evaluate SpMV against CPU and GPU and compare the graph algorithm implementations to Ligra [172]. Accelerators are specifically optimized for certain applications by eliminating extraneous hardware overhead for programmability and flexibility and thus are not considered for performance and energy efficiency analysis for a fair comparison.

### 4.5.2.1 Standalone SpMV

Figure 4.8 demonstrates the speedup and energy efficiency gain of SpMV, on a suite of real-world graphs, over CPU and GPU implementations. Overall, CoSPARSE achieves an average speedup of  $4.5\times$  and  $17.3\times$  compared to the CPU and GPU, respectively. Although the GPU has a significantly higher core count and peak memory bandwidth compared to the CPU, the irregular and low-locality memory accesses, coupled with the thread divergence inherent in the SIMT model, bottleneck the GPU. Memory dependence stalls account for 32% of the GPU stalls (increasing with vector density), and most of the remaining cycles (averaging 35%) are spent in synchronization, instruction fetching, and throttled memory accesses. Despite the high memory bandwidth, the highest average bandwidth utilized by a kernel varies from 12-71%, and the overall performance is  $<0.006\%$  of the peak performance. The CPU shows better performance than the GPU because the out-of-order cores can hide the overhead of the irregular memory accesses and handle complex execution flow. High power consumption is observed in both the CPU and GPU because of the massive number of threads in the GPU and the high-performance out-of-order cores

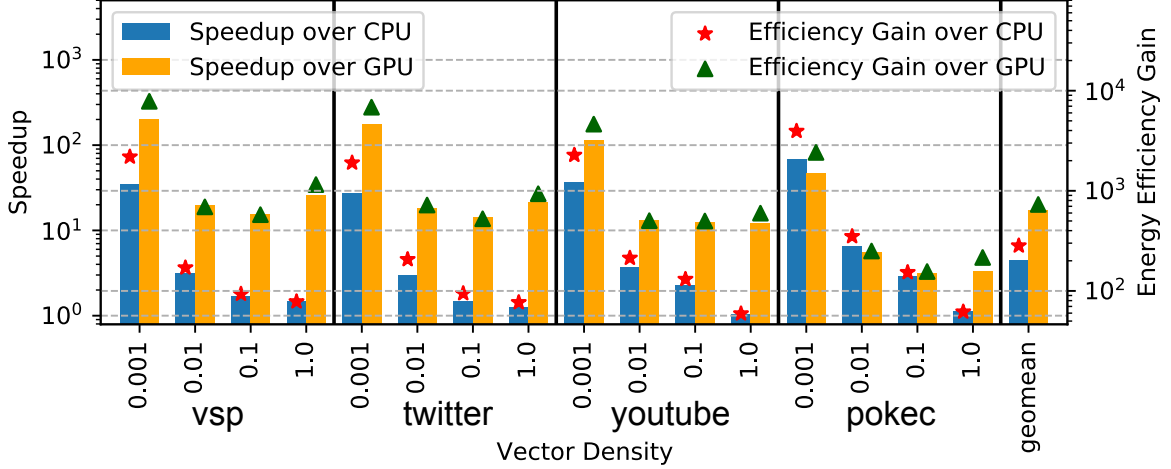


Figure 4.8: Speedup and energy efficiency gain of CoSPARSE ( $16 \times 16$ ) over CPU and GPU. The vector density sweeps from 0.001 to 1.0. CoSPARSE achieves an average speedup (energy efficiency gain) of  $4.5 \times (282.5 \times)$  and  $17.3 \times (730.6 \times)$  over CPU and GPU, respectively.

in the CPU. In contrast, the underlying architecture of CoSPARSE uses lightweight in-order cores and a flexible memory hierarchy. CoSPARSE improves memory parallelism and locality by determining the best software and hardware configuration, and carefully scheduling and balancing the workload. The average energy efficiency gain over CPU and GPU are  $282.5 \times$  and  $730.6 \times$ , respectively.

The performance and energy efficiency gains grow as the vector becomes sparser, since CoSPARSE takes advantage of the vector sparsity and skips computations and accesses to the output vector if the vector element is zero. For vectors with densities lower than 0.01, the underlying algorithm switches from inner product to outer product (except for soc-pokec), and further eliminates accesses to matrix elements that correspond to zero elements in the vector. Since soc-pokec has the largest dimension, it has more columns to merge for the same vector density, and thus, outer product only performs better than inner product for a vector density of 0.001.

#### 4.5.2.2 Graph Analytics Algorithms

We first conduct a case study illustrating the execution of graph analytics on our CoSPARSE framework. Figure 4.9 shows the execution time per iteration running SSSP with soc-pokec normalized to inner product in the Trans-SC mode. From *Iter 4* to *Iter 8*, the inner product implementation outperforms outer product because of the high vector density (as large as 47% in *Iter 6*). Within these inner product iterations, *Iter 6* and *Iter 7* have the highest vector density and achieve the best performance in the Trans-SCS mode, whereas *Iter 4*, *Iter 5*,



and 8 favor the Trans-SC mode. The rest of the iterations involve vector densities less than 0.5% and achieve better performance using outer product in the Trans-PC mode. The synergistic software and hardware reconfiguration amass a net speedup of  $1.51\times$ , over the Trans-SC-only inner product execution, *i.e.* a baseline implementation with no software or hardware reconfiguration. Similar trends are observed with BFS and SSSP for the rest of the graphs. The combined software and hardware reconfiguration achieves a speedup of up to  $2.0\times$  across different algorithms and input graphs.

Iteration	Vector Density	Normalized Execution Time					Best Configuration	
		Inner Product		Outer Product			SW	HW
		SC	SCS	SC	PC	PS		
0	<1%	1.0	1.0	<0.1	<0.1*	<0.1	OP	PC
1	<1%	1.0	1.1	<0.1	<0.1*	<0.1	OP	PC
2	<1%	1.0	1.2	0.1	0.1*	0.1	OP	PC
3	<1%	1.0	1.2	0.6	0.5*	0.6	OP	PC
4	1%	1.0*	1.2	7.5	6.7	6.8	IP	SC
5	12%	1.0*	1.1	>10	>10	>10	IP	SC
6	47%	1.0	0.8*	>10	>10	>10	IP	SCS
7	27%	1.0	0.9*	>10	>10	>10	IP	SCS
8	5%	1.0*	1.0	4.1	3.7	3.8	IP	SC
9	<1%	1.0	1.1	0.5	0.4*	0.4	OP	PC
10	<1%	1.0	1.0	0.1	0.1*	0.1	OP	PC
11	<1%	1.0	1.0	<0.1	<0.1*	<0.1	OP	PC
12	<1%	1.0	1.1	<0.1	<0.1*	<0.1	OP	PC
13	<1%	1.0	1.1	<0.1	<0.1*	<0.1	OP	PC


 Reconfiguration
 \* The execution time of the best configuration

Figure 4.9: Vector density, execution time normalized to inner product in Trans-SC, and the best software/hardware configuration for each iteration of CoSPARSE ( $16\times 16$ ) for SSSP on soc-pokec. Each iteration is color coded with the best configuration. The best configuration changes with the active vertex set, which conforms to the analysis in Sec. 4.3.3.

The performance and energy efficiency gain of CoSPARSE on a  $16\times 16$  system over Ligra on a Xeon CPU is shown in Figure 4.10. In terms of performance, CoSPARSE outperforms Ligra in most cases and achieves a maximum speedup of  $3.5\times$ . Ligra outperforms CoSPARSE for soc-pokec on BFS and SSSP slightly because the CPU has much more hardware resources, *e.g.* on-chip memory, to handle the large memory footprint of soc-pokec. However, the CPU consumes at least  $200\times$  more power and  $40\times$  more area than CoSPARSE. Upon normalizing the performance by the power consumption, we obtain an efficiency gain of  $84\times$  for BFS and  $129\times$  for SSSP. Overall, CoSPARSE achieves

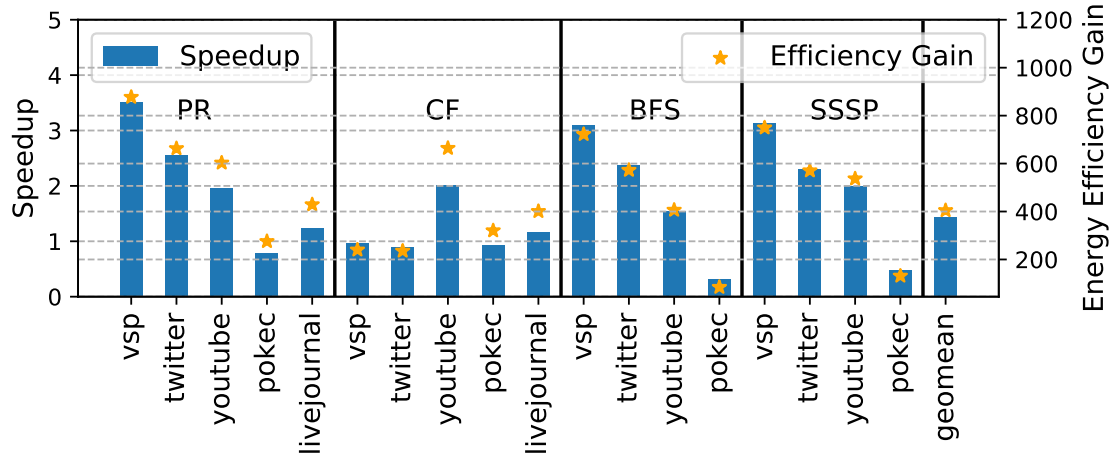


Figure 4.10: Speedup and efficiency gain of CoSPARSE ( $16 \times 16$ ) over Ligra (Intel Xeon E7-4860 at 2.6 GHz, 48 cores with 256GB DRAM).

an average energy efficiency gain of  $404.4 \times$  across all evaluated algorithms and graphs, compared to Ligra.

## 4.6 Conclusion

This work proposed CoSPARSE as a novel solution that combines software and hardware reconfiguration strategies to optimize the performance and efficiency of SpMV, and thereby SpMV-based graph analytics algorithms. We mapped different SpMV algorithms with custom scheduling and workload balancing onto an architecture with fast reconfiguration of the on-chip memory hierarchy. In contrast to other established heuristics, which require user input for configuration decisions, CoSPARSE is a fully-automated system that judiciously decides the best-performing software/hardware configuration. The parameters that guide the reconfiguration decision-making engine are obtained by evaluating SpMV on a wide range of matrices and system sizes. CoSPARSE is evaluated against existing platforms, for both SpMV and a selection of common graph analytics algorithms. For SpMV, CoSPARSE showed significant speedups ( $4.5 \times$  and  $17.3 \times$  on average) and energy-efficiency gains ( $282.5 \times$  and  $730.6 \times$  on average) compared to the CPU and GPU, respectively. CoSPARSE also provides an energy efficient platform for graph analytics; compared to Ligra on CPU, CoSPARSE achieved an average speedup and energy efficiency improvement of  $1.5 \times$  and  $404.4 \times$ , respectively.

## CHAPTER 5

# Near-Memory Multi-way Merge Solution for Sparse Data Merging

The design process of CoSPARSE helps us discover a challenge that was neglected but has become increasingly important in recent years, *i.e.* the matrix (graph) transposition overhead. (Sec. 4.3.4.2). In CoSPARSE and many state-of-the-art graph processing frameworks, different matrix formats are stored simultaneously in the main memory for fast algorithm reconfiguration by avoiding the transposition overhead on-the-fly, trading off storage for performance. Sparse matrix transposition is also widely used in preprocessing. Though many prior works claim that the overhead can be amortized through repeated execution, as we step into the big data era, the dataset size has exploded to a certain level that the preprocessing cost has become no longer negligible. As an important building block of sparse linear algebra applications, sparse matrix transposition has received much less attention than many other sparse kernels, such as SpMM and SpMV.

This chapter presents MeNDA, a near-memory multi-way merge solution for sparse matrix transposition and general sparse data merging. NMP has been extensively studied to optimize memory-intensive workloads. However, none of the proposed designs address sparse matrix transposition. Prior work shows that sparse matrix transposition does not scale as well as other sparse primitives such as SpMV and hence has become a growing bottleneck in common applications. Sparse matrix transposition is highly memory-intensive but low in computational intensity, making it a promising candidate for NMP. Our proposed solution, MeNDA, is a scalable near-DRAM multi-way merge accelerator that eliminates the off-chip memory interface bottleneck and exposes the high internal memory bandwidth to improve performance and reduce energy consumption for sparse matrix transposition. MeNDA adopts a merge sort based algorithm, exploiting spatial locality, and proposes a near-memory PU featuring a high-performance hardware merge tree. Because of the wide application of merge sort in sparse linear algebra, MeNDA is an extensible solution that can be easily adapted to support other sparse primitives such as SpMV. Techniques includ-

ing seamless back-to-back merge sort, stall-reducing prefetching, and request coalescing are further explored to take full advantage of the increased system memory bandwidth. Compared to two state-of-the-art implementations of sparse matrix transposition on a CPU and a sparse library on a GPU, MeNDA is able to achieve a speedup of  $19.1\times$ ,  $12.0\times$ , and  $7.7\times$ , respectively. MeNDA also shows an efficiency gain of  $3.8\times$  over a recently proposed SpMV accelerator integrated with HBM. Incurring a power consumption of only 78.6 mW, a MeNDA PU can be easily accommodated by commodity Dual In-line Memory Modules (DIMMs).

The work presented in this chapter was published as a paper at ISCA'22 [52].

## 5.1 Introduction

As a fundamental primitive in many important application domains such as graph analytics, machine learning, and scientific computation [160, 69, 23, 192, 151, 80, 214, 211, 175, 54, 10], Sparse linear algebra applications are notoriously memory-intensive due to the irregular memory access pattern. Recently, there has been a surge in customizing hardware accelerators near memory to tackle sparse linear algebra applications such as sparse gathering [8, 97, 117], SpMV [8, 165, 200], and graph analytics [37, 144, 213, 218, 5]. However, none of these works address sparse matrix transposition.

Sparse matrix transposition converts a sparse matrix stored in the column-major order to the row-major order or vice versa. It is an essential building block for a wide range of applications, such as biconjugate gradient [58], standard quasi-minimal residual [59], and algebraic multigrid methods [204]. In addition, many recent graph analytics frameworks adaptively switch between different representations of the dataflow, which requires either frequent sparse matrix transposition on-the-fly or multiple copies of the input graph in different orders [172].

Merge sort is a common approach for sparse matrix transposition [195]. Some recent NMP proposals implement outer product based SpMV by adopting a reduction tree to merge sort the partial columns [165, 8]. But these designs targeting SpMV cannot perform sparse matrix transposition for two reasons. First, reduction trees in these systems usually perform merge sort based on the row indices of matrix elements and thus do not care about the order of the column indices, while sparse matrix transposition needs to take into account the order of both indices. Second, unlike SpMV, which outputs a dense vector, sparse matrix transposition outputs a sparse matrix, which is irregular and requires a much higher output bandwidth. To support sparse matrix transposition, all these issues need to be addressed.

In contrast to many other sparse primitives, sparse matrix transposition involves no arithmetic operations but integer comparisons to reorder the nonzero elements. Therefore, the performance of sparse matrix transposition highly depends on the attainable memory bandwidth. However, the effective system bandwidth that can be utilized by transposition is restricted both by the theoretical peak bandwidth that the memory interface can provide and the contention at the memory interface, which is confirmed by the roofline model and the scalability analysis presented in Sec. 5.2.2. All these constraints make sparse matrix transposition a promising candidate for NMP because NMP exposes the high internal memory bandwidth of memory devices and avoids the contention bottleneck at the memory interface. Instead of integrating accelerators with 3D/2.5D-stacked memory devices, in this paper, we focus on a DIMM-based design for its cost-efficient capacity scaling, which is critical for workloads involving large datasets.

Designing a near-DRAM solution for sparse matrix transposition poses four unique challenges. First, for lack of reduction, sparse matrix transposition requires high bandwidth for both input fetching and output streaming. Hence, sending the output directly to the host like prior sparse gathering proposals [97, 8] is not feasible. Second, due to the large dataset size and the limited on-chip storage, recent CPU implementations [195] for transposition transfer intermediate data back-and-forth between the host and the main memory, exhibiting a significant amount of memory traffic. Because near-DRAM accelerators have more strict area restrictions and consequently even less area for Static Random Access Memory (SRAM), reducing the amount of intermediate data transfer is more difficult. Third, performing parallel transposition on multiple concurrent PUs is non-trivial. To exploit the high internal bandwidth, accelerators are usually employed in the buffer chip of a DIMM beside each rank. Thus communications across PUs in different DIMMs need to go through the off-chip memory interface, which is prohibitively expensive and can easily become the performance bottleneck [180]. Finally, near-memory transposition puts additional requirements on the data layout. The transposition process should not change the data representation and should allow easy access to the matrix NZEs as the standard compressed formats after transposition. These requirements together make designing an efficient and scalable PU with minimal modifications to the commodity DIMM hardware a challenging task.

To tackle these challenges, we propose MeNDA, a scalable NMP solution for sparse matrix transposition. The key component of MeNDA is a lightweight PU featuring a hardware merge tree deployed in the buffer chip of a DIMM. The merge tree is designed to be very wide to reduce the number of merge sort iterations, which is proportional to the amount of intermediate data transfer, and supports seamless execution of multiple rounds of

merge sort to minimize stalls in execution. Techniques including stall-reducing prefetching and request coalescing are also explored to further improve memory bandwidth utilization. MeNDA proposes a novel data layout to avoid communications between PUs and keep a consistent compressed format for both the input and output matrix, enabling a software-agnostic transposition backend. The data layout also considers workload balancing to maximize parallelism and memory bandwidth utilization.

Merge sort is widely employed in sparse linear algebra applications, making MeNDA an efficient solution for many sparse dataflows. Finally, to showcase its applicability to other sparse dataflows, we illustrate how MeNDA can be adapted to perform SpMV, which is a fundamental kernel for machine learning and graph analytics [54, 165, 8, 151].

Specifically, we make the following contributions:

1. An in-depth characterization of sparse matrix transposition which unveils the memory-bound nature and the request contention bottleneck at the memory interface, motivating the adoption of NMP.
2. A scalable NMP solution for sparse matrix transposition, MeNDA, which explores DIMM- and rank-level parallelism by placing custom PUs beside each DRAM rank. The PUs feature lightweight hardware merge trees and are enhanced with techniques including seamless back-to-back merge sort, stall-reducing prefetching, request coalescing, and workload balancing to fully utilize the exposed high internal memory bandwidth.
3. Adaptation of MeNDA to SpMV, demonstrating that MeNDA is an extensible and efficient solution to multi-way merge dataflows in sparse linear algebra applications.
4. A heterogeneous programming model to completely hide the implementation details of MeNDA from end users and enhance ease of adoption.
5. Qualitative and quantitative analyses of the benefits and overhead of integrating MeNDA into existing designs for sparse linear algebra applications.

MeNDA is an efficient solution that can be easily integrated into the buffer chip of a commodity DIMM and is evaluated against state-of-the-art implementations on CPU and GPU on a suite of real-world sparse matrices. . Experiments show that MeNDA achieves an average speedup of  $19.1\times$  and  $12.0\times$  over `scanTrans` and `mergeTrans` on CPU, respectively, and  $7.7\times$  over `cuSPARSE` on GPU. Compared to a recent near-memory SpMV accelerator based on HBM, MeNDA shows an efficiency gain of  $3.8\times$ .

## 5.2 Background and Motivation

Sparse matrix transposition is widely used in sparse linear algebra applications but has received much less attention than many other sparse kernels, such as SpMM and SpMV [195]. Based on the roofline model and the thread scaling analysis, sparse matrix transposition can potentially achieve great performance benefits and energy savings from NMP since it has low computational intensity while being heavily memory bandwidth bound.

### 5.2.1 Sparse Matrix Formats and Sparse Matrix Transposition

Sparse matrices are often stored in compressed formats to save storage and avoid computations on zero elements. Commonly used formats are Compressed Sparse Row Format (CSR) and CSC. As shown in Figure 5.1, CSR/CSC stores a sparse matrix in three arrays: (1) an index array for the column(/row) index of each NZE, (2) a value array for the value of each NZE, and (3) a pointer array for the start pointer of NZEs of each row(/column).

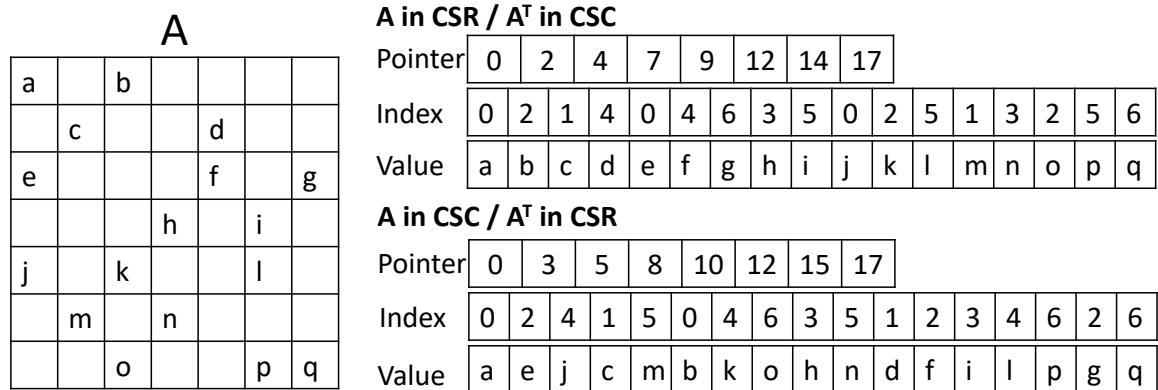


Figure 5.1: Transposition and compressed storage formats for sparse matrices.

Sparse matrix transposition transforms a  $M \times N$  sparse matrix  $A$  to a  $N \times M$  matrix  $A^T$  by swapping the row index and column index of each NZE. Therefore, transposing a sparse matrix is in essence equivalent to converting a sparse matrix from the CSR format to the CSC format, or the opposite. As can be seen from Figure 5.1, the CSC representation of a sparse matrix  $A$  is equivalent to the CSR representation of its transpose  $A^T$ . For simplicity, we will use converting a matrix from CSR to CSC to denote general sparse matrix transposition from this point of the dissertation.

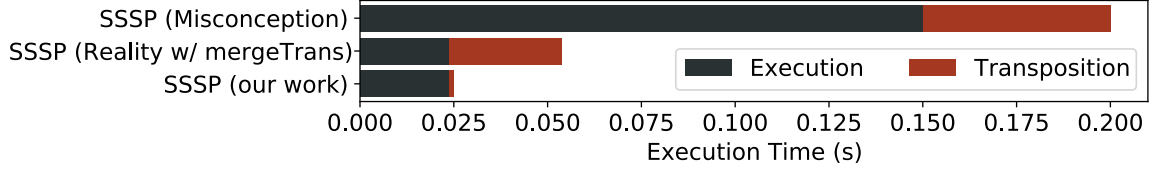
Sparse matrix transposition is an essential building block in both the processing and preprocessing stages of sparse linear algebra applications [195]. Typical examples are linear system solvers such as biconjugate gradient [58] and standard quasi-minimal resid-

ual [59]. Despite the fact that considering the scenario of consuming a fixed sparse matrix, the overhead of preprocessing (including sparse matrix transposition) can be amortized by iterative execution, many recent works have shown that this overhead is becoming no longer negligible as the dataset size grows and have taken this overhead into account in the evaluation [136, 208]. There are also applications that are not iterative enough to amortize the transposition overhead or have to transpose a changed sparse matrix each iteration. For example, the simultaneous localization and mapping problem requires a new information matrix at each step, and performing  $A^T A$  on the new matrix dominates the execution time [41, 123].

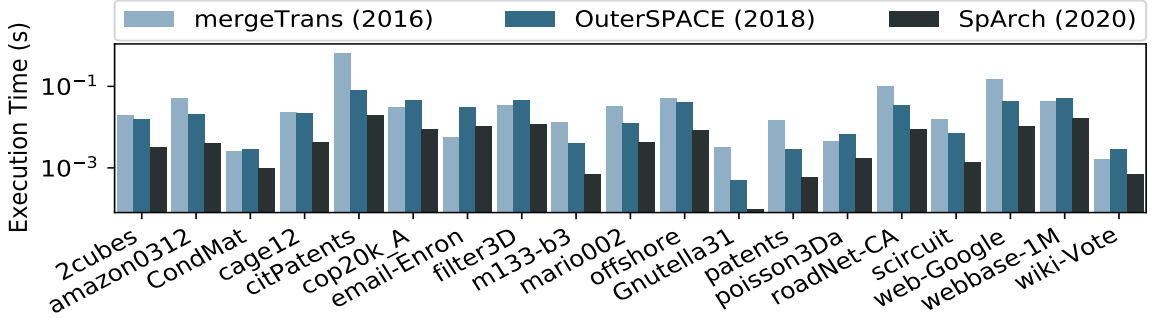
Since Beamer *et al.* [17] first proposed a hybrid approach for BFS, many recent graph analytics frameworks have built upon this work and adopted dynamic reconfiguration between a sparse and a dense representation of the dataflow based on the active vertex set [30, 38, 54, 71, 132, 145, 172, 194, 196, 212, 217]. The dynamic reconfiguration greatly improves performance but requires the original graph  $A$  for one representation and its transpose  $A^T$  for the other representation during execution. A common misconception regarding the transposition overhead is shown in the top bar in Figure 5.2(a), *i.e.* the transposition overhead is minor compared to the execution time of an end-to-end workload and can be easily amortized. However, the reality (middle bar in Figure 5.2(a)) is that recent breakthroughs in algorithms and architectures have significantly improved the performance of graph processing. Consequently, runtime transposition using a state-of-the-art implementation [195] can introduce a 126% performance overhead to a recently proposed graph framework [54]. Therefore, graph frameworks usually store more than one copy of the input graph in different formats to avoid the performance overhead of transposing the graph on-the-fly.

Although many recent efforts have been spent on optimizing sparse primitives, sparse matrix transposition has not received as much attention. As shown in Figure 5.2(b), the execution time of SpMM has been improved from being comparable to that of sparse matrix transposition (OuterSPACE, 2018 [151]) to being much less than that of transposition (SpArch, 2020 [214]). These efforts only further increase the percentage of time taken by sparse matrix transposition in a workload, making it a more noteworthy bottleneck. Therefore, coming up with an efficient solution for sparse matrix transposition has become increasingly important.





a) Execution breakdown of Single Source Shortest Path (SSSP) with runtime transposition



b) Execution time of transposition(mergeTrans) and SpMM(OuterSPACE/SpArch)

Figure 5.2: (a) Breakdown of SSSP execution time on CoSPARSE [54] for graph amazon based on common misconceptions, using mergeTrans[195], and using our work, MeNDA. (b) Execution time comparison of recent proposals for sparse matrix transposition (mergeTrans) and SpMM (OuterSPACE[151] / SpArch[214]). Recent hardware breakthroughs have greatly optimized sparse applications, *e.g.* SpMM and SpMV, whereas little research effort has been spent on accelerating sparse matrix transposition, making transposition an increasingly evident bottleneck.

## 5.2.2 Characterizations on Sparse Matrix Transposition

To understand the bottleneck of sparse matrix transposition, we performed characterizations on mergeTrans [195], a merge sort based sparse matrix transposition implementation on CPUs. The methodology for these experiments is detailed in Sec. 5.5.

### 5.2.2.1 Roofline Analysis

A roofline mode [199] of sparse matrix transposition is built and presented in Figure 5.3(a). The throughput is measured through the NNZ generated per second (NNZ/s), which is a metric introduced in [153]. The roofline model shows that sparse matrix transposition lies in the memory-bound region. Specifically, the throughput achieved is within only 25% of the theoretical maximum and bottlenecked by the system memory bandwidth. The impact of exposing the high internal memory bandwidth on throughput is revealed by lifting the roofline by  $8\times$  [97]. The throughput is improved by  $4.1\text{-}5.2\times$ , which shows the potential benefit of applying NMP on sparse matrix transposition. Meanwhile, sparse matrix transposition has much lower computational intensity than common sparse routines such as SpMM and SpMV because no floating point operations are involved. *The high mem-*

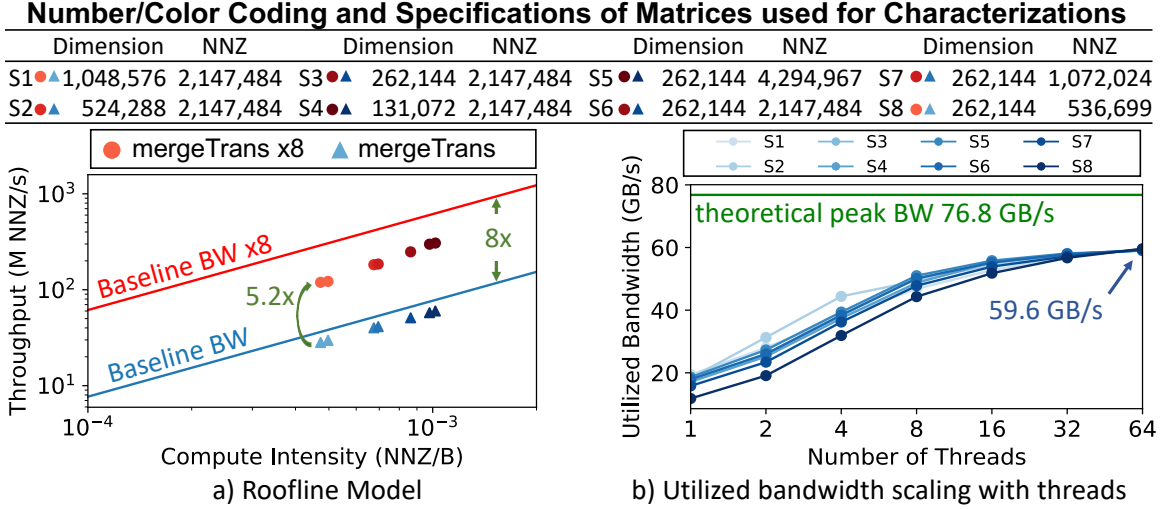


Figure 5.3: (a) Roofline model of `mergeTrans` [195] running with 64 threads. Sparse matrix transposition is memory bandwidth bound because the data points are close to the "roof", *i.e.* the red and blue lines that label the peak throughputs which can be achieved when the system memory bandwidth is fully utilized. (b) Memory bandwidth utilized by `mergeTrans` with an increasing number of threads. The memory bandwidth utilization saturates before reaching maximum due to the bottleneck at the memory interface.

*ory requirement and low arithmetic intensity make sparse matrix transposition a promising candidate for NMP [33].*

### 5.2.2.2 Thread Scaling Analysis

Prior work shows that the performance of state-of-the-art sparse matrix transposition implementations does not scale well with the increasing number of threads [195]. To further analyze the scalability of sparse matrix transposition, we measured the utilized bandwidth with an increasing number of threads, as shown in Figure 5.3(b). While the theoretical peak bandwidth, represented by the green horizontal line, is 76.8 GB/s, the achievable maximum bandwidth is around 62 GB/s [97]. In Figure 5.3(b), the utilized memory bandwidth starts to saturate at 16 threads and reaches the maximum at 64 threads at 59.6 GB/s. In practice, little performance benefit is observed beyond 16 threads and further bandwidth saturation is undesirable due to significantly increased memory latency. *What efficient sparse matrix transposition will most benefit from is an approach that reduces memory latency and relieves the contention at the off-chip memory interface by avoiding transferring data back-and-forth between the host and the memory device.*

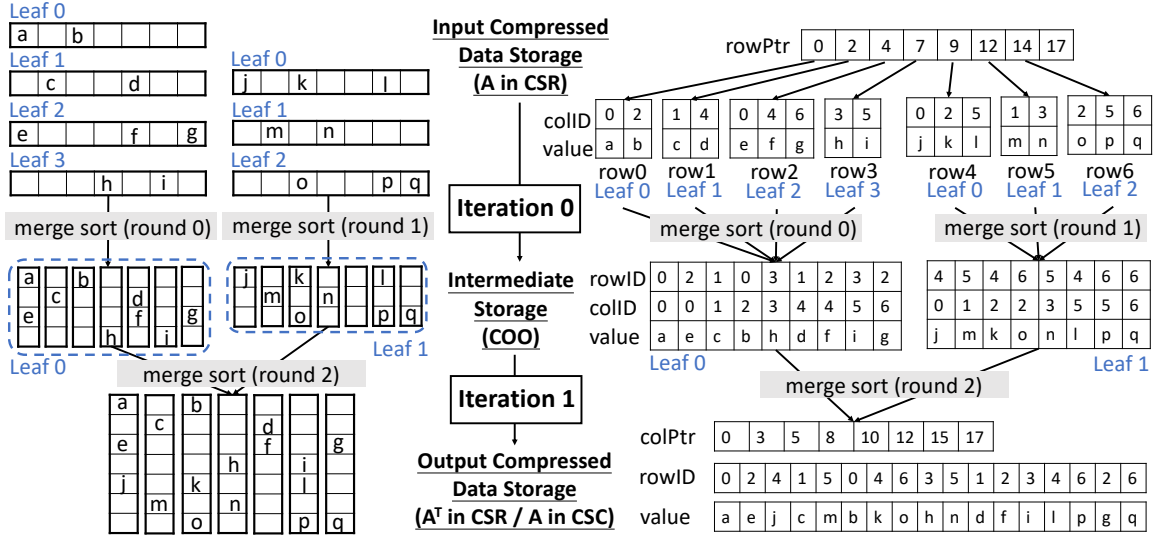


Figure 5.4: Dataflow of MeNDA performing transposition on the sparse matrix in Figure 5.1. Each round of merge sort is executed sequentially on a 4-way merge tree. Left: The outcome of each round in the dense data structure. Right: The real input and output data of each round that are stored in memory. The input and output data are stored in the compressed data storage formats (CSR/CSC), and the intermediate data are stored in COO.

### 5.3 MeNDA System Architecture

Prior work proposed two algorithms for parallel sparse matrix transposition - a count sort based algorithm (`scanTrans`) and a merge sort based algorithm (`mergeTrans`) [195]. In this work, we adopted the merge sort algorithm not only because merge sort presents higher spatial locality but also because merge sort is widely used in sparse linear algebra [151, 54, 165]. Inspired by prior near-DRAM accelerators [97, 11], the near-memory PUs are embedded in the buffer chips of DIMMs to minimize the modifications to commodity DRAM devices. The proposed solution is scalable as a higher throughput can be achieved by populating a memory channel with multiple MeNDA-enabled DIMMs. To take full advantage of the exposed high internal memory bandwidth, the custom PU features a very wide multi-way merge tree supporting seamless back-to-back merge sort, stall-reducing prefetching, and request coalescing. To further improve parallelism, a novel data layout is proposed to eliminate communications and balance workloads among PUs. As mentioned in Section 5.2, the design details of MeNDA are explained using the paradigm of converting a sparse matrix from the CSR format to the CSC format.

### 5.3.1 Algorithm and Dataflow

MeNDA applies the merge sort algorithm to perform sparse matrix transposition. Figure 5.4 demonstrates the dataflow of transposing the matrix in Figure 5.1 using a 4-leaf hardware merge tree. An  $l$ -leaf merge tree merges  $l$  incoming sorted streams into a single sorted stream in a round. Since the 4-leaf merge tree does not have enough hardware resources to merge sort all matrix rows, more than one iteration is needed. As shown in Figure 5.4, in iteration 0, the first four rows and the last three rows are merged subsequently. Then in iteration 1, the two sorted streams are merged into the final output. In practice, the number of iterations required to finish transposition equals  $\log_l N$ , where  $l$  refers to the number of leaves in the merge tree and  $N$  refers to the number of non-empty matrix rows.

The input and output data are both stored in the compressed format, *i.e.* the input in CSR and the output in CSC. If the algorithm needs more than one iteration to finish, the intermediate data are stored in the COO format. COO stores the row index, column index, and value of each NZE in three separate arrays so that accesses to the intermediate data can exploit bank-level parallelism. Due to matrix sparsity, an intermediate sorted stream may contain numerous empty rows/columns. Therefore, COO tends to take up less storage than CSR/CSC and is also easier to decode. The memory space for the input sorted streams is freed immediately after they are processed. Therefore, a runtime storage overhead of  $O(l \cdot N)$  is required, where  $l \ll N$ . In contrast, storing a second copy of the matrix requires an overhead of  $O(N^2)$ .

### 5.3.2 Processing Unit (PU) Microarchitecture

MeNDA places PUs in the data buffer chips of DIMMs beside each rank to minimize modifications to DRAM devices and to explore DIMM- and rank-level parallelism. Each PU concurrently transposes a partition of the matrix and issues memory requests to the corresponding ranks in parallel. The effective memory bandwidth available to MeNDA thus scales with the total number of ranks.

A MeNDA PU consists of a merge tree, prefetch buffers, a controller, a request queue, and a memory interface unit (Figure 5.5). In the merge tree, each PE is connected to two child PEs through a FIFO unless it is a leaf node. An  $l$ -leaf merge tree thus has  $l - 1$  PEs and  $\log_2 l$  levels, *i.e.* at least  $\log_2 l$  cycles are required for data to travel from a leaf PE to the root PE. The existence of FIFOs allows each PE to pop one data packet every cycle without a critical path from the root PE to the leaf PEs. The root PE is connected to an output buffer, which allows store requests to be sent at memory block granularity (64B). Each leaf PE is connected to two prefetch buffers through FIFOs. Prefetch buffers are in

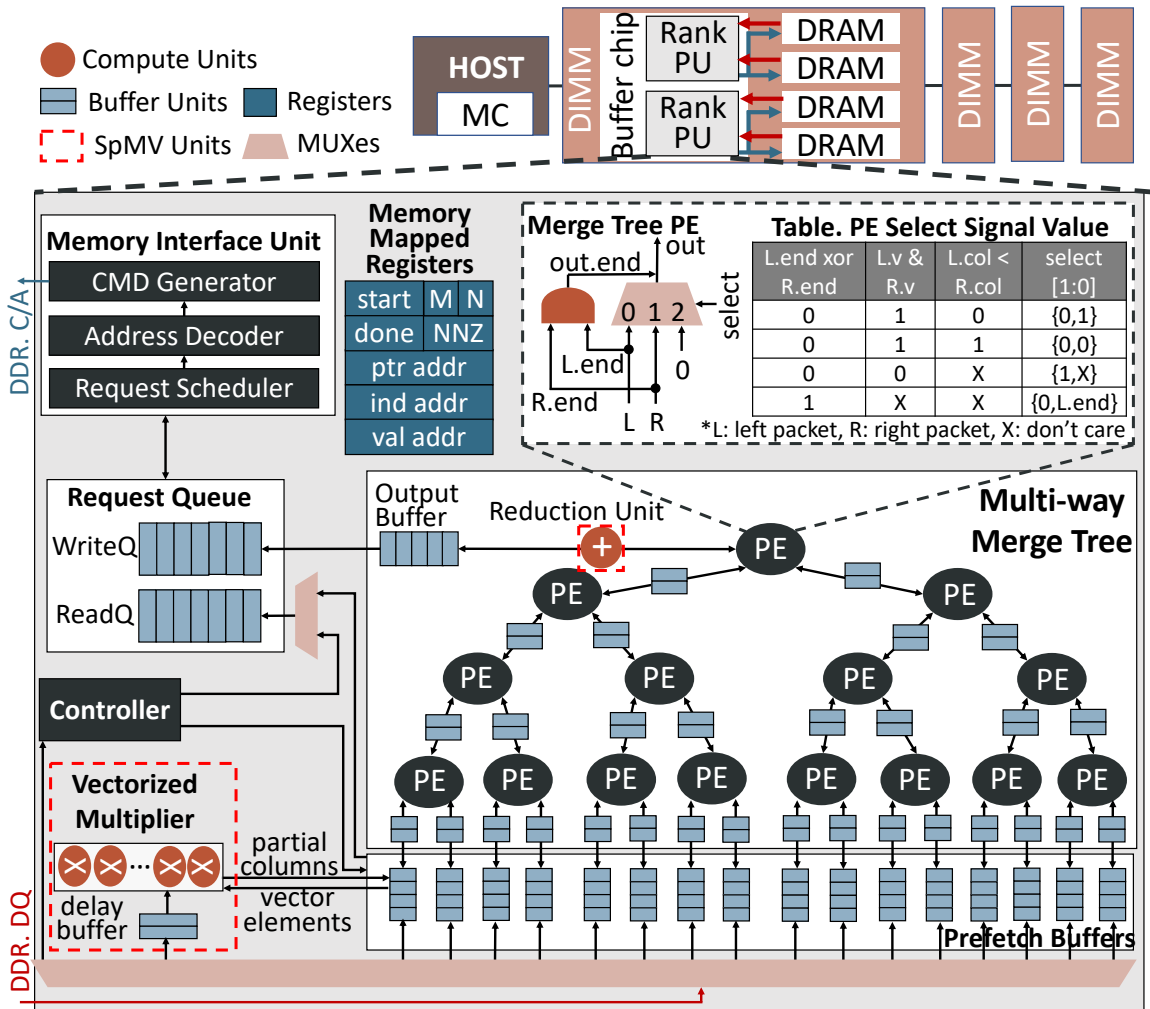


Figure 5.5: Architecture of MeNDA (top) and a MeNDA PU (bottom). A PU consists of a merge tree, prefetch buffers, a controller, a request queue, and a memory interface unit. The extra units required to support SpMV, *i.e.* a delay buffer and floating point adders and multipliers, are highlighted in red boxes.

charge of sending memory load requests and feeding the leaves with correct data. The controller is a Finite-State Machine (FSM) that assigns each prefetch buffer the start and end addresses of the corresponding sorted streams. Theoretically, in each cycle, only one load request is sent to the prefetch buffers because only one element is popped from the root PE. Similarly, only one store request is sent to the prefetch buffers to fill in the data from the memory bus because only one memory response can return each cycle. Therefore, to reduce power consumption, the prefetch buffers are implemented as multi-bank SRAM. The design goal of the merge tree is to saturate the internal memory bandwidth while fitting in the buffer chip, which, according to the evaluation, is satisfied by the current design.

Data are transferred among PEs through data packets containing a 1-bit valid signal and the 32-bit row index, the 32-bit column index, and the 32-bit value of a NZE. Only when both child PEs provide valid packets will a PE pop the data packet with the smaller column index and send the packet to its parent PE or the output buffer if it is the root PE. All the memory requests are sent to a request queue with separate queues for loads and stores and processed by a memory interface unit, which mimics a simplified memory controller. The memory interface unit consists of a request scheduler that selects the request with the highest priority from the request queue, an address decoder that translates the incoming physical address to a DRAM address, and a command generator that generates DRAM commands for the chosen request. The request scheduler selects requests based on a First Come First Serve - First Ready (FCFS-FR) policy that prioritizes requests ready to launch and requests resulting in DRAM row hits.

### **5.3.3 Seamless Back-to-back Merge Sort**

Real-world sparse matrices tend to be extremely large and sparse, causing each iteration of sparse matrix transposition, especially the first iteration, to handle many rounds of merge sort of short input streams. Hence, it is important to reduce the stalls between different rounds of merge sort. An end-of-line signal is added to the data packet to signify the end of a sorted stream and allow seamless execution of multiple rounds of merge sort. The prefetch buffer sets the end-of-line signal when the last element of a sorted stream is sent. The PEs propagate the end-of-line signal when both child PEs set the end-of-line signal. Instead of starting a new round of merge sort after the current round of merge sort has finished, the prefetch buffers feed their PEs with data for the next round immediately after the end-of-line signal is set.

Figure 5.6 illustrates the seamless execution of the first and second rounds of merge sort in Figure 5.4. Load requests for the second round of merge sort are sent in cycle 1,

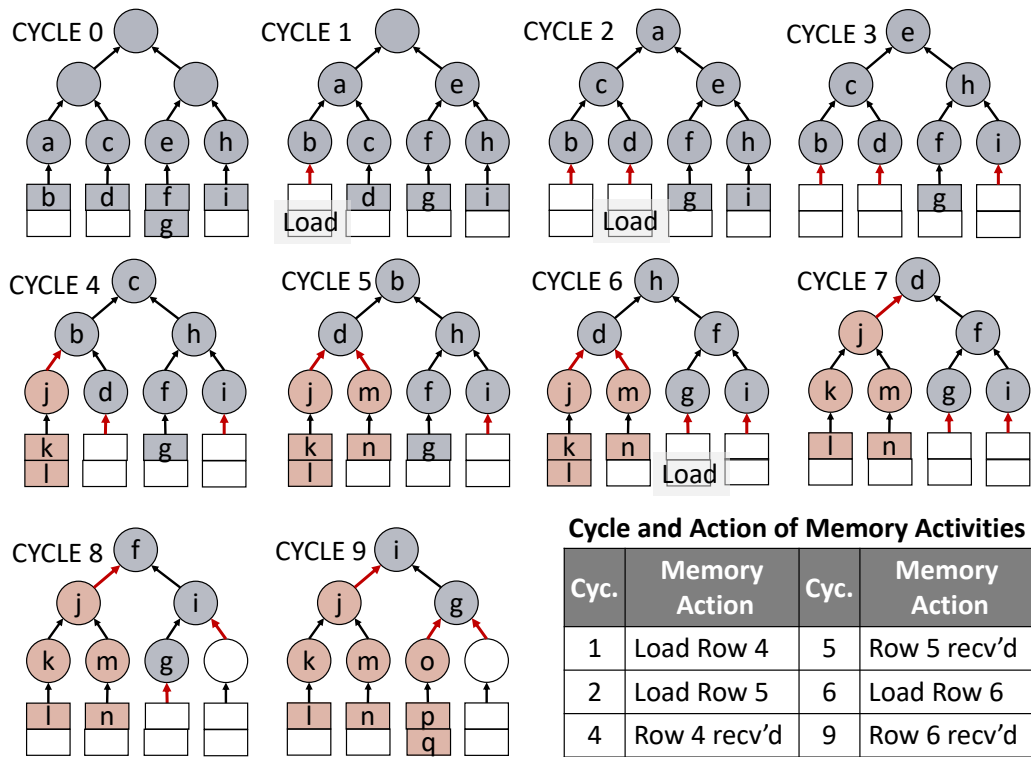


Figure 5.6: Timing diagram of data propagation for merge sort shown in Figure 5.1 on a 4-leaf merge tree assuming a memory latency of 3 cycles. The cycle number and the corresponding memory activities are shown in the bottom right table. End-of-line signal propagation is shown with red arrows.

2, and 6, *i.e.* as soon as the prefetch buffers become empty. Propagating the end-of-line signals enables the merge tree to produce effective results without stalls. If the merge sort is executed one after another, in the scenario presented in Figure 5.6, the first round of the merge sort ends at cycle 10 and then the three load requests for the second round are sent. The second round of the merge sort is not able to start until cycle 15 due to memory stalls, and the merge tree thus remains idle for 5 cycles. The use of the end-of-line signals not only maximizes the hardware resource utilization but also helps distribute burst memory requests at the start of a new round of merge sort evenly over time.

### 5.3.4 Memory Bandwidth Utilization Optimizations

The prefetch buffers aim to make the best use of the fetched memory blocks and reduce merge tree stalls. However, even launching load requests as soon as the prefetch buffers become empty would cause the merge tree to stall while waiting for the memory responses. Therefore, **stall-reducing prefetching** is proposed so that memory load requests are sent whenever a prefetch buffer can fit the requested data. Assuming a prefetch buffer can fit

16 NZEs and 4 NZEs have been popped to the leaf PE, if the number of NZEs left in the current sorted stream is less than or equal to 4, the memory requests for the subsequent NZEs will be issued. However, a prefetch buffer is not allowed to send memory requests for more NZEs when there are outstanding memory requests even if the prefetch buffer can accommodate the NZEs. This is because, to reduce merge tree stalls, it is more desirable to keep all prefetch buffers non-empty than serially filling each prefetch buffer until full.

While stall-reducing prefetching aims at taking full advantage of the available memory bandwidth, **request coalescing** is designed to reduce the total memory traffic. Due to matrix sparsity, multiple matrix rows can be co-located in the same memory block. In this case, memory load requests for the same memory block can be sent from different prefetch buffers in the first iteration. Request coalescing avoids sending these duplicate memory requests to the memory device by checking the read request queue each time a new load request is enqueued. If a load request to the same memory block is found, the incoming request will be merged into the same request queue slot. Since the memory response is broadcast to all the prefetch buffers, merging the duplicate memory requests does not affect the functional correctness of the design and there is no need to keep track of the requesters. Because the prefetch buffers are implemented as multi-bank SRAM and the prefetch buffers that send the same memory requests are usually neighbors, the memory response from a merged request can fill multiple prefetch buffers in one cycle by interleaving neighboring prefetch buffers to different SRAM banks. Minimal additional hardware is required to support request coalescing. Specifically, a comparator is added to each entry of the read request queue to enable parallel address matching, similar to a Content-Addressable Memory (CAM). Synthesis of the RTL model shows that the additional hardware has negligible impact on the frequency and the area of PUs.

Taking the example in Figure 5.6, if row 6 of the input matrix has only one element  $o$ , stall reducing prefetching allows the load request for  $o$  to be issued in cycle 1 instead of cycle 6. Request coalescing merges this request into the prior request for row 4, making  $o$  available in cycle 4 instead of cycle 9.

### 5.3.5 Input Operand Co-location and Workload Balancing

In near-DRAM accelerators, communications between PUs, especially those across DIMMs, need to go through the off-chip memory interface and thus are prohibitively expensive. A common challenge is to keep all the input operands local in a single rank for a rank-level PU [33]. To avoid communications between MeNDA PUs, each PU is assigned a contiguous chunk of the sparse matrix, *i.e.* each PU is responsible for transposing a horizontal



partition of the input sparse matrix. The original CSR format can then be directly used without preprocessing, and it is also easy to locate a NZE after transposition.

A naïve way to partition the sparse matrix is to use the Most Significant Bits (MSBs) of the address to assign NZEs to a rank. However, this could cause severe workload imbalance. For example, assuming a total of 8 ranks, if the 3 MSBs of the input array range from 000 to 100, only rank 0 to rank 4 will be assigned work while rank 5 to rank 7 remain idle throughout the execution. Since the execution time of a PU is roughly proportional to the NNZ assigned to it, an NNZ-based partitioning technique is desired.

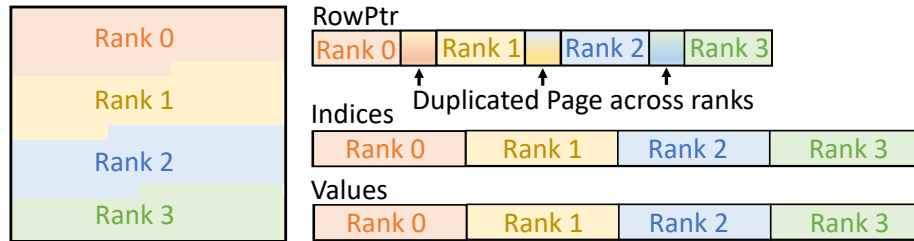


Figure 5.7: Matrix partitioning across 4 ranks.

The workload balancing takes place during data allocation using the technique proposed in [33]. The host first uses the number of MeNDA PUs and the NNZ of the input matrix to determine the NNZ assigned to each PU and then allocates contiguous chunks of physical memory accordingly. To ensure that the index and value of each NZE assigned to a PU are mapped to the corresponding rank, page coloring is used to specify the rank a physical page belongs to, and thus the data assigned to a PU needs to be aligned by page. However, the same technique does not apply to the row pointer array because the rank that a row pointer belongs to depends on the matrix distribution. Therefore, the host needs to calculate the start and end row indices of the NZEs assigned to a PU and then assign the corresponding pages of the row pointer array to the target rank using page coloring. In the case that one page of the row pointer array is needed by two ranks, the page will be duplicated and each rank will have a private copy, leading to a maximum total storage overhead of  $page\_size \times \#ranks$ , which is negligible for typical datasets. Figure 5.7 shows a partitioned sparse matrix given 4 ranks. The start and end addresses of the row pointer, index, and value arrays of each rank are written to specific memory-mapped registers for PUs to calculate the target addresses during computation.

### 5.3.6 Adaptation to SpMV

Merge sort is widely used in sparse linear algebra applications. A typical example is outer product based SpMV. The merge phase of SpMV has the same dataflow as sparse matrix

transposition and thus can be implemented directly on MeNDA. As transposition does not involve floating point computations, to support SpMV, a reduction unit consisting of three pipelined floating point adders is inserted between the root PE and the output buffer. In addition, a vectorized floating point multiplier is placed next to the prefetch buffers. The additional hardware units required to support SpMV are highlighted with red rectangles in Figure 5.5. When executing sparse matrix transposition, these units will be gated and incur no power overhead.

The input matrix is stored in a partitioned CSC format, which matches the format of the transposed matrix generated by our work. The reason to apply horizontal partitioning to the input matrix is that each PU would generate a partition of the final vector instead of a partial result vector. Due to the irregular distribution of sparse matrices, the horizontal matrix partition processed by a PU can have numerous empty columns. To reduce the memory loads to the pointers and vector elements that correspond to the empty columns, an auxiliary pointer array is constructed to label the memory blocks in the pointer array that contain non-empty columns.

Each time the controller sends a load request for the column pointers based on the auxiliary array, it also issues a request to fetch the vector elements that need to be multiplied with these columns. In contrast to sparse matrix transposition, the column indices are not needed for computation because all columns are eventually merged into a single vector. Hence, the space in the prefetch buffers aimed to store the column indices for transposition is now reused to store the vector elements instead. When a read request for the matrix values returns, the data is sent to the multiplier. Meanwhile, the prefetch buffers that are waiting for this memory response snoop the memory bus and send the stored vector elements to the multiplier. However, the needed vector elements could be unavailable at the moment because the load request for the vector elements is still outstanding. This is very likely due to request reordering caused by the scheduling policy and request coalescing. To deal with this situation, a delay buffer is designed to register the response and notify the request scheduler to prioritize requests for vector elements until the request needed by the registered response is served. The outputs of the multiplier are broadcasted and stored into the prefetch buffers. Note that the multiplication is only performed in the first iteration, *i.e.* the multiplier is disabled starting from the second iteration. When an element with the smallest row index is popped from the root PE, the root PE compares its index with prior outputs and merges the elements with the same index using the reduction unit. The intermediate vectors are stored in (index, value) pairs, and the output vector is stored in a dense array.

## 5.4 Programming Model and Interface

MeNDA adopts a heterogeneous programming model, similar to prior NMP proposals [97, 11]. The host is responsible for memory allocation and initialization for tasks offloaded to PUs. Figure 5.8(a) shows the pseudo-code of a sample graph analytics workload based on the CoSPARSE implementation [54]. In line 0-2, the host performs memory allocation and workload balancing partitioning as described in Sec. 5.3.5 for the input sparse matrix. The allocation functions also write the necessary metadata to the corresponding memory-mapped registers. The host can access the allocated data structures with no modifications to the original implementation because the allocation functions have taken care of the virtual to physical address mapping, which is hidden from the host.

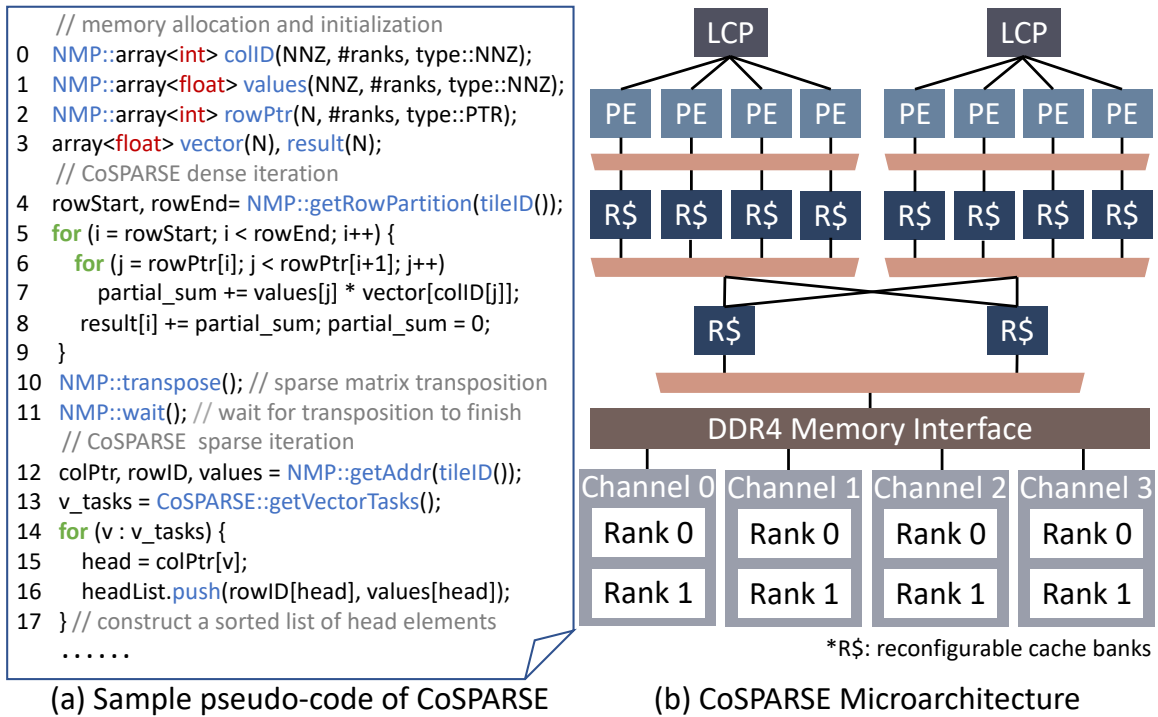


Figure 5.8: (a) Sample pseudo-code of CoSPARSE using the programming interface of MeNDA and (b) the microarchitecture of the hardware substrate of CoSPARSE with 2 processing tiles and 4 PEs per tile.

In line 10, the host launches the sparse matrix transposition through a non-blocking function call `NMP::transpose()`, which sets the start signals of PUs by writing to the memory-mapped registers. While the PUs are transposing the matrix, the host can concurrently execute other kernels. Prior work has proposed techniques to efficiently allow concurrent accesses from both the host and NMP PUs [33]. However, it is still undesirable for the host to execute memory-intensive workloads because sparse matrix transposition

is already heavily memory bandwidth bound. Since sparse matrix transposition can easily saturate the memory bandwidth, executing another memory-intensive workload on the host will only severely hurt the performance of both tasks.

Upon finishing transposition, a PU sets the finish signal and updates the addresses of the transposed matrix in the memory-mapped registers. In the case that the transposed matrix is required for subsequent code execution, `NMP::wait()` can be used to block the host execution until the transposition finishes, as shown in line 11. `NMP::wait()` is implemented similarly to a conditional variable, which gets notified to resume the host execution as soon as the finish signals of all the PUs are set. After transposition, each rank will hold a horizontal partition of the sparse matrix stored in CSC. To access the data in a column, `NMP::getAddr(i)` is used to obtain the start addresses of the data arrays in rank  $i$  (line 12).

### 5.4.1 Integrating MeNDA with Existing Platforms

The programming interface of MeNDA aims at minimizing the modifications to the standard compressed storage format of sparse matrices so that minimal code changes are required to integrate MeNDA. The potential performance overhead of integrating MeNDA comes in two ways. First, the proposed data layout assigns each rank with a contiguous chunk of the sparse matrix with the same NNZ. This requires modifications to the address mapping and support from the page table of the operating system. Second, after transposition, the sparse matrix is stored in multiple horizontal partitions in CSC, which needs the host implementation to adapt to the partitioned data storage. To access an entire column, the host needs to access the sub-column in each rank.

To analyze the performance overhead, we implemented MeNDA on CoSPARSE [54], a recent graph analytics framework on a reconfigurable hardware substrate [152]. An architecture overview of CoSPARSE is shown in Figure 5.8(b). CoSPARSE performs SpMV in inner product using row-major COO for the dense iterations, and outer product using CSC for the sparse iterations. To apply MeNDA, the dense iteration implementations are the same except that the memory address mapping is different. For the sparse iterations, since CoSPARSE uses preprocessing that performs horizontal partitioning based on NNZ, CoSPARSE can directly use the post-transposition data format and save preprocessing overhead with minor modifications to the implementation. Assuming a CoSPARSE system of  $A$  tiles and  $B$  PEs per tile and where there are  $R$  DRAM ranks in total, for simplicity, we let tile  $A/R \times i$  to  $A/R \times (i + 1) - 1$  work on the horizontal partition in rank  $i$ .

Many recent designs use NNZ-based partitioning [54, 165] and thus similar implemen-

Table 5.1: Parameters of Ramulator and MeNDA.

Ramulator CPU Parameters					
L1	32KB	L2	256KB	LLC	3MB
Cache	64B block size, 8-way associative, 16 MSHR entries				
Ramulator DRAM Parameters					
Standard	DDR4_2400R				
Organization	4Gb_x8				
Scheduling	32-entry RD/WR queue, FRFCFS_PriorHit				
Timing Parameters	tRC=55, tRCD=16, tCL=16, tRP=16, tBL=4, tCCDS=4, tCCDL=6, tRRDS=4, tRRDL=6, tFAW=26				
Processing Unit Parameters					
Frequency	800 MHz	Number of Leaves		1024	
No. FIFO Entry	2	No. Prefetch Buffer Entry		32	
No. Read/Write Queue Entry				32	
FP Units (SpMV only)		16 3-stage FP Mult, 3 2-stage FP Add			

tations can apply. Even if the host needs to access each DRAM rank to access and process a column, for graph analytics workloads, the sparse iterations access only a small subset of columns, and the dense iterations usually take up the majority of the total execution time (Figure 5.11). Therefore, there are many use cases that would benefit from MeNDA without introducing a significant performance overhead.

## 5.5 Experimental Methodology

This section details the experimental methodology that is used to characterize `mergeTrans` and evaluate MeNDA.

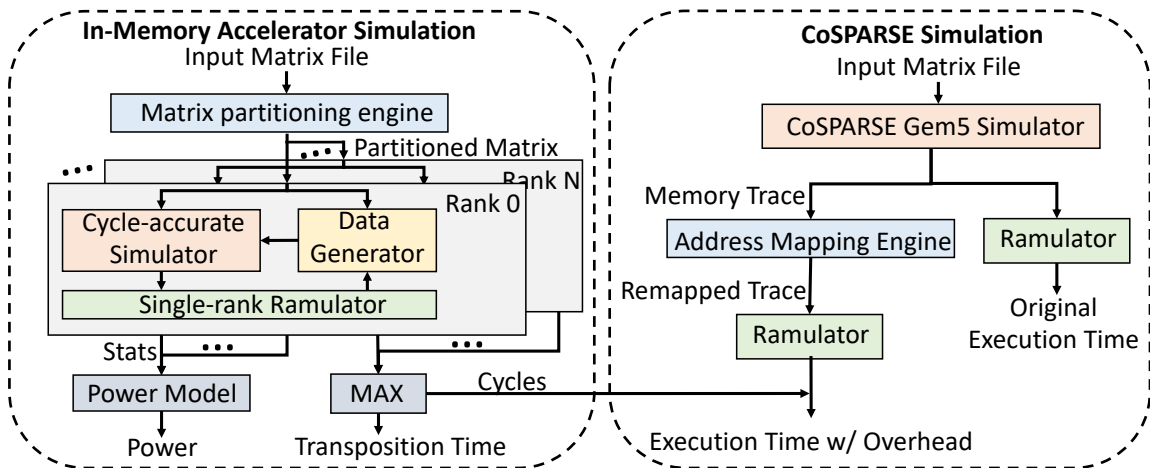


Figure 5.9: Experimental methodology for MeNDA.

Table 5.2: Specifications of CPU and GPU baselines.

Platform	Specifications
CPU	AMD Ryzen Threadripper 2990WX, 32 cores/64 threads at 3.0-4.2 GHz, 128 GB DDR4 memory @ 68.3 GB/s, 213 mm <sup>2</sup> (12 nm)
GPU	NVIDIA Tesla V100, 5120 CUDA cores at 1.25 GHz, 16 GB HBM2 memory at 900 GB/s, 815 mm <sup>2</sup> (12 nm)

### 5.5.1 Simulation Methodology

To model the performance of MeNDA, we designed a cycle-accurate simulator and connected the memory interface to Ramulator [106], as shown in Figure 5.9 (left). The system parameters are shown in Table 5.1. The area and power estimations are based on the synthesis of an RTL model of the PU in 40nm using Synopsys design compiler.

**Characterizations on mergeTrans** The roofline model and the thread scaling analysis (Figure 5.3) are built through trace simulation of mergeTrans[195] on Ramulator. We created a trace generator that collects the memory trace and ran the traces in `cpu` mode of Ramulator with a custom implementation of barrier synchronization to improve simulation accuracy. The parameters used in Ramulator are shown in Table 5.1.

**Integration with CoSPARSE** The performance impact of integrating MeNDA is estimated on CoSPARSE [54] assuming a system size of  $8 \times 16$ , *i.e.* 8 tiles with 16 PEs per tile. As shown in Figure 5.9 (right), the memory trace is collected using the gem5 simulator [20] and then processed by a memory re-mapping engine based on the strategy described in Sec. 5.3.5. Both the original and the re-mapped memory trace are then executed on Ramulator in `dram` mode to obtain the performance of CoSPARSE after integrating MeNDA.

### 5.5.2 Baseline and Benchmarks

We evaluate MeNDA against `scanTrans` and `mergeTrans` from [195] on the CPU and `cusparseCsr2cscEx2` from cuSPARSE v11.4.0 on the GPU. The specifications of the CPU and GPU are detailed in Table 5.2. The CPU and GPU power are measured using AMDuProf and `nvidia-smi`, respectively. The specifications of the evaluated synthetic and real-world matrices are shown in Table 5.3 and Table 5.4, respectively. The power-law matrices are generated using SNAP RMat generator GenRMat. The real-world matrices are selected from the SuiteSparse Matrix Collection [40].

Table 5.3: Specifications of Synthetic Uniform\* (N#) and Power-law†(p#) Matrices.

Matrix	Dimension	NNZ	Matrix	Dimension	NNZ
N1/P1	262,144	3,435,973	N5/P5	524,288	8,388,608
N2/P2	262,144	1,717,986	N6/P6	1,048,576	8,388,608
N3/P3	262,144	858,993	N7/P7	2,097,152	8,388,608
N4/P4	262,144	429,496	N8/P8	4,194,304	8,388,608

\*Generated by randomly sampling NZEs until NNZ is reached.

†Generated using GenRMat (Dimension, NNZ, 0.1, 0.2, 0.3) (snap.py).

Table 5.4: Specifications of SuiteSparse Matrices [40].

Matrix Dimension,NNZ Kind	Plot	Matrix Dimension,NNZ Kind	Plot	Matrix Dimension,NNZ Kind	Plot
amazon 262K,1.23M Directed graph		ASIC_320K 321K,1.93M Circuit simulation		bcsstk32 44K,2.01M Structural problem	
language 399K,1.22M Directed graph		mac_econ 206K,1.27M Economic problem		parabolic 525K,3.67M Fluid dynamics	
rajat21 411K,1.88M Circuit simulation		sme3Dc 43K,3.15M Structural problem		Slashdot0902 82K,948K Directed graph	
stomach 213K,3.02M 2D/3D problem		transient 178K,961K Circuit simulation		twotone 120K,1.21M Circuit simulation	
venkat01 62K,1.72M Fluid Dynamics		webbase-1M 1.00M,3.11M Directed graph		wiki-Talk 2.39M,5.02M Directed graph	

## 5.6 Evaluation

This section evaluates the performance, area, and power of MeNDA for sparse matrix transposition and SpMV. In addition, the benefits of integrating MeNDA with existing designs and the optimizations proposed in Sec. 5.3.4 are presented. Finally, the performance impact of the matrix properties and the system size and frequency on MeNDA are studied.

### 5.6.1 Comparison with CPU and GPU Baselines

MeNDA is compared to state-of-the-art sparse matrix transposition implementations on CPU and GPU in Figure 5.10. *The speedup of MeNDA over baselines comes from both the reduction in memory traffic and the improvement in memory bandwidth utilization.*

Taking `wiki-Talk` as an example, compared to `mergeTrans`, MeNDA reduces the memory traffic by  $11.2\times$  while exhibiting  $2.7\times$  higher bandwidth utilization. These result from both the exposed high internal memory bandwidth and the optimizations in Sec. 5.3.4. *In general, MeNDA achieves higher throughput on large, less sparse matrices.* MeNDA performs better on less sparse matrices because less memory bandwidth is then spent on accessing and updating the pointer array, which does not contribute to the throughput, which is measured in NNZ/s. In the case that the number of iterations to finish transposition remains the same, MeNDA favors larger matrices as bank-level parallelism can be better exploited when there are more sorted streams to merge in the last iteration. `mergeTrans` and `scanTrans`, however, do not scale as well for large, sparse matrices, and perform the worst on `wiki-Talk`. Accordingly, MeNDA shows the most speedup over `mergeTrans` and `scanTrans` on this matrix.

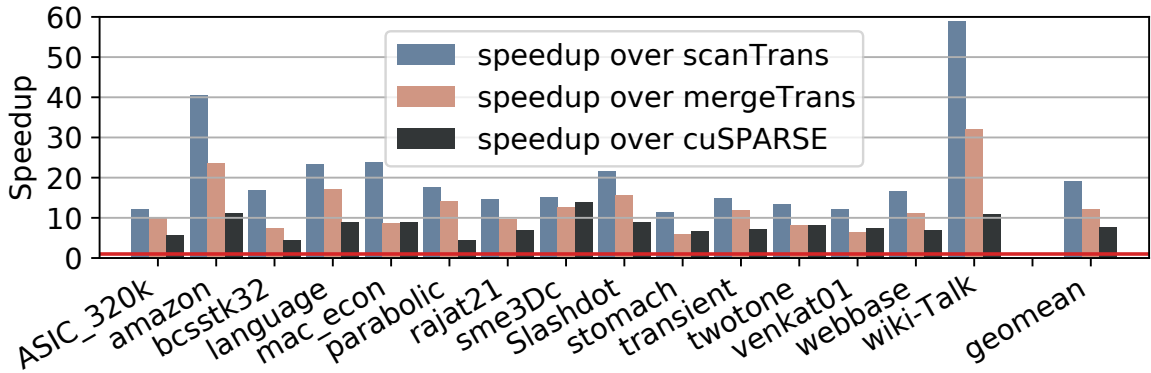


Figure 5.10: Speedup of MeNDA over `scanTrans` and `mergeTrans` on CPU [195] and `cuSPARSE` on GPU. The red line labels the speedup of 1.

The performance of `cuSPARSE` also favors less sparse matrices and is sensitive to matrix distribution. `bcsstk32` and `sme3dc` have similar dimensions and densities, but the throughput of `cuSPARSE` on `bcsstk32` is much higher than `sme3dc`. Because the performance of MeNDA is not affected by matrix distribution, which is further proved in Sec. 5.6.6, MeNDA achieves the highest speedup over `cuSPARSE` on `sme3dc` and the lowest speedup for `bcsstk32`. *Overall, MeNDA achieves an average speedup of  $19.1\times$ ,  $12.0\times$ , and  $7.7\times$  compared to `scanTrans`, `mergeTrans`, and `cuSPARSE`, respectively.*

## 5.6.2 Area and Power Analysis

A MeNDA PU consumes 78.6 mW at 800 MHz and takes up  $7.1 \text{ mm}^2$  in 40 nm. The extra logic required to support SpMV adds negligible area and up to 13.8 mW power consump-



tion. Given the estimations of prior works [97, 11] and that a typical data buffer chip takes up  $100 \text{ mm}^2$  [138], the PU is within the power constraint and can be integrated into the buffer chip of a DIMM, introducing a small area and power overhead.

### 5.6.3 Benefits and Overhead Analysis on End-to-end Workloads

To analyze the performance benefits and overhead of integrating MeNDA into existing designs, the execution time of CoSPARSE performing SSSP algorithm on the graph `amazon` with and without MeNDA is illustrated in Figure 5.11. Though the number of the sparse iterations is twice that of the dense iterations, the majority (87%) of execution time is taken up by the dense iterations. The potential performance overhead of MeNDA comes from two sources – the additional execution time due to the memory mapping required by MeNDA and the execution time of the transposition.

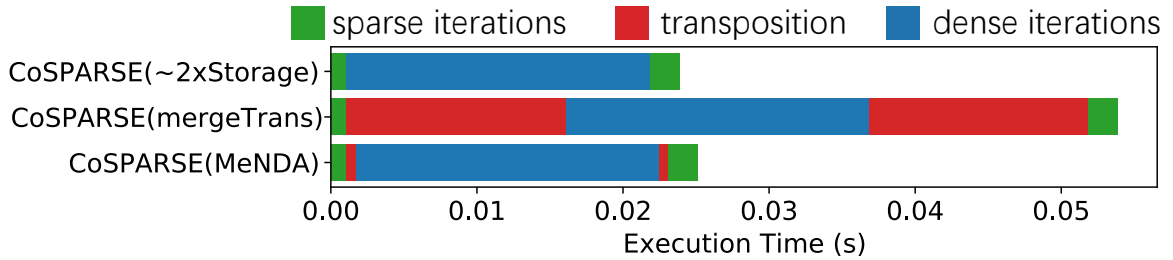


Figure 5.11: Execution time of SSSP on CoSPARSE for `amazon` without runtime transposition, with runtime transposition using `mergeTrans`, and with runtime transposition using MeNDA. CoSPARSE ( $\sim 2x$ Storage) avoids runtime transposition at the cost of storing two copies of the graph [54].

Although integrating MeNDA requires the matrix partition assigned to a PU to reside in a rank, as shown in Figure 5.11, the change in memory mapping has a negligible impact on the execution time of the SSSP algorithm. This is because the PEs in CoSPARSE work on all matrix partitions concurrently to exploit memory-level parallelism, resulting in all the DRAM ranks being accessed in parallel. Therefore, rank-level parallelism is still well exploited. Sparse matrix transposition is launched each time CoSPARSE switches from the dense dataflow to the sparse dataflow or the opposite. In practice, sparse matrix transposition is commonly performed at most twice for a graph algorithm execution. As shown in Figure 5.11, integrating MeNDA for dynamic matrix transposition decreases the transposition overhead from 126% to 5% while allowing CoSPARSE to store only one copy of the graph in DRAM, reducing the required storage by almost half, thus supporting a larger graph within a fixed DRAM size. As dataset sizes keep growing, MeNDA can prevent designs like CoSPARSE from expensive disk accesses when the DRAM devices

can only fit a single copy of the graph, at the cost of a minor transposition latency.

### 5.6.4 Memory Bandwidth Utilization Optimization Analysis

The execution time of MeNDA with different optimizations enabled and prefetch buffer sizes is shown in Figure 5.12. A key observation is that *request coalescing greatly benefits the first iteration by reducing total memory traffic while stall-reducing prefetching improves the performance of the following iterations by increasing memory bandwidth utilization.*

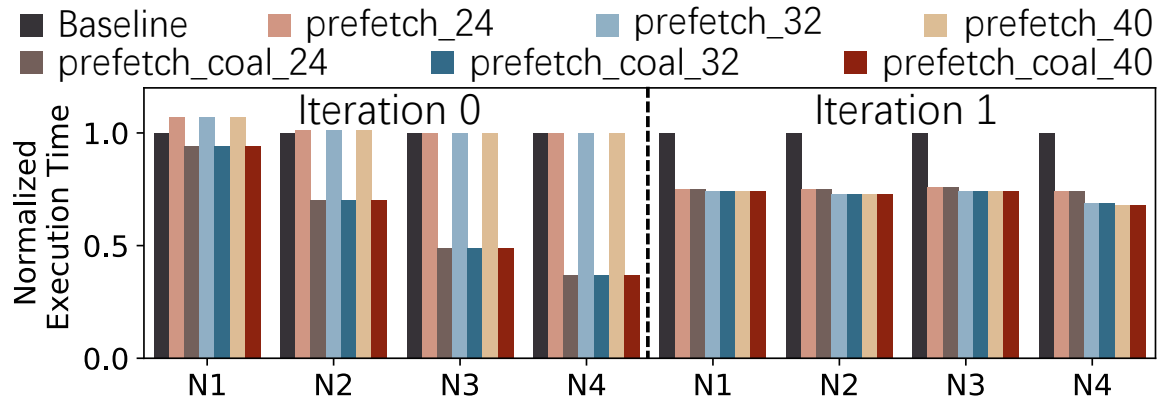


Figure 5.12: The execution time of MeNDA applying different optimizations normalized to that of the baseline implementation. In the legend, "prefetch" refers to stall-reducing prefetching enabled, "coal" refers to request coalescing enabled, and the number refers to the size of the prefetch buffers.

Stall-reducing prefetching fetches data needed in the future in advance to keep the prefetch buffers non-empty and thus reduce the stalls of the merge tree. Although stall-reducing prefetching has little impact on the total amount of memory traffic, it improves the memory bandwidth utilization by 8-16%, leading to 12-16% better performance. Larger prefetch buffers enable the merge tree to send out more prefetch requests. However, little performance improvement is seen after the size of the prefetch buffer reaches 32. This is because the memory bandwidth is already saturated and the prefetch buffers are not able to send out more requests even if there are vacancies. This is also demonstrated by the fact that, when request coalescing is not enabled, stall-reducing prefetching can sometimes worsen the performance of the first iteration. The reason is that the excessive prefetching requests block the critical read requests on demand, resulting in performance degradation.

Request coalescing, instead, benefits the first iteration much more than the other iterations, especially for sparser matrices. Because sparser matrices have fewer NZEs per row, *i.e.* each memory block can accommodate more rows, a single memory response can fill more prefetch buffers. On the other hand, after the first iteration, sorted streams are usu-

ally much longer than a memory block, so there is little opportunity for request coalescing. Therefore, the following iterations barely benefit from request coalescing. Experiments show that request coalescing reduces the memory traffic of iteration 0 by up to 60%, leading to a maximum speedup of  $2\times$ . Overall, stall-reducing prefetching and request merging can achieve a speedup of  $1.2\times$  to  $2.1\times$  compared to a baseline with no optimizations.

### 5.6.5 Scalability Analysis

MeNDA places PUs at DRAM rank level, and thus the performance scales with the number of ranks. In the synthetic matrices,  $N1 - N4$  have the same matrix dimensions but decreasing densities while  $N5 - N8$  have the same NNZs but increasing matrix dimensions. As shown in Figure 5.13, the throughput of MeNDA scales almost linearly with the increasing number of channels. The execution time of transposing  $N1$  to  $N4$  decreases with NNZ while that of  $N5$  to  $N8$  remains similar. The throughput of MeNDA decreases slightly from  $N1$  to  $N4$  and from  $N5$  to  $N8$  under a fixed number of channels. This is because when the size of the pointer array increases with the matrix dimension and becomes even larger compared to the index and value array, accessing and updating the pointer array takes up a larger portion of the memory bandwidth usage. However, this does not contribute to the throughput, which is defined as NNZ/s, and thus results in a throughput degradation. In summary, the throughput of MeNDA is proportional to the total number of ranks, and the execution time scales with the NNZ of the input matrix, assuming the number of iterations in the execution is fixed.

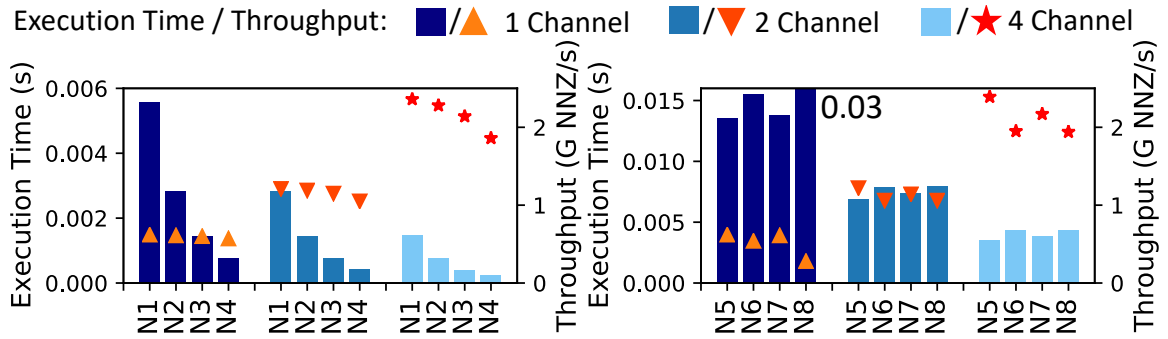


Figure 5.13: Execution time and throughput of MeNDA sweeping matrix size and density and the number of channels.

Transposing  $N8$  on one channel is an outlier because  $N8$  is the largest synthetic matrix and requires three iterations to finish while all other matrices finish within two iterations. Adding an iteration to the execution significantly increases the total memory traffic and severely degrades the throughput. Therefore, it is desirable to minimize the number of iter-

ations in the execution. In this work, the nominal number of leaf PEs is 1024, which allows transposition to be finished within two iterations for matrices with a size up to  $1024^2 \times R$ , where  $R$  is the total number of DRAM ranks.

### 5.6.6 Matrix Distribution Analysis

Many real-world matrices have irregular distributions, especially those in the graph analytics domain. However, Figure 5.14 shows that the performance of MeNDA is barely affected by matrix distribution. *Although in most cases, the power-law matrices take longer to transpose, the differences in execution time remain within 10%.* This can be attributed to the workload balancing strategy (Sec. 5.3.5), which divides tasks evenly among PUs to improve parallelism, and the seamless back-to-back merge sort feature (Sec. 5.3.3), which maximizes hardware resource utilization.

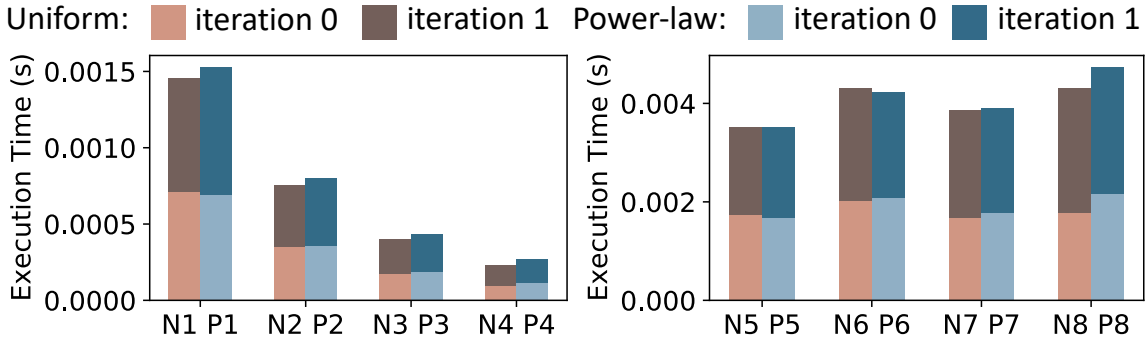


Figure 5.14: The execution time of the uniform matrices compared with that of the power-law matrices with the same sizes and densities.

### 5.6.7 Design Space Exploration

Figure 5.15 (left) presents the execution time and EDP of MeNDA under different frequencies. *Because MeNDA already saturates the memory bandwidth, increasing the system frequency beyond 800 MHz brings little performance benefit and simply boosts the power consumption, resulting in a higher EDP.* Although 600 MHz presents a lower EDP, this work prioritizes performance and selects 800 MHz as the nominal frequency. In a scenario where EDP is the most important metric, a lower frequency can be used at the cost of performance.

The execution time and EDP of MeNDA with merge trees of different sizes are shown in Figure 5.15 (right). The size of the merge tree does not affect the throughput but impacts the number of iterations needed to finish the sparse matrix transposition. A PU with a

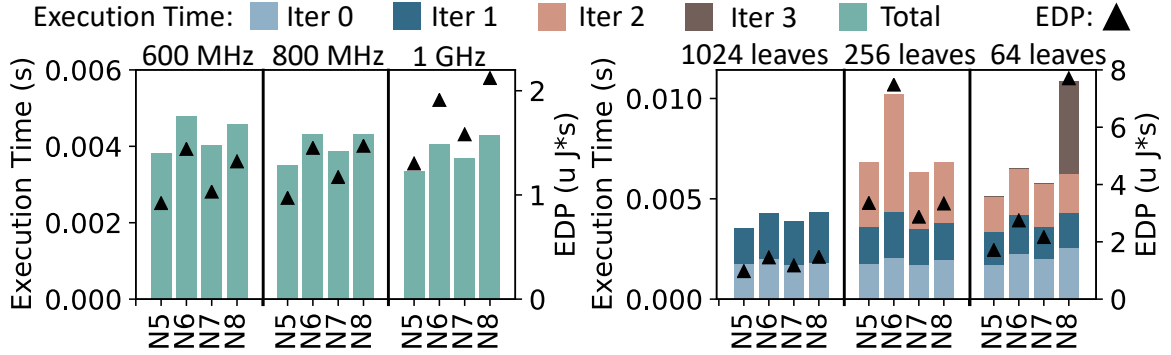


Figure 5.15: The execution time and EDP of MeNDA sweeping the accelerator frequency (left) and number of leaf PEs (right).

1024-leaf merge tree can transpose  $N5$  to  $N8$  in two iterations. With 256 leaves, three iterations are needed. With only 64 leaves,  $N5$  to  $N7$  can still finish in three iterations but  $N8$  requires four iterations. The reduction in power consumption resulting from using a merge tree with fewer leaf PEs does not offset the performance degradation caused by the increase in the number of iterations. Hence, *the PU with a 1024-leaf merge tree has not only the best performance but also the lowest EDP.*

The execution time of  $N6$  is much longer than that of the other matrices on a 256-leaf merge tree. This is because  $N6$  does not have enough rows that the third iteration only merges two sorted streams, and loading the two sorted streams induces many row conflicts. Although  $N5$  has an even lower number of rows and the third iteration has at most two sorted streams, the majority of the NZEs reside in one of the sorted streams. Therefore spatial locality is well exploited when loading the long sorted stream.  $N7$  and  $N8$ , on the other hand, have much more rows than  $N6$  and thus have more sorted streams to merge in the third iteration. The percentage of row conflicts in the third iteration is 57% for  $N6$  but 43% for  $N7$ . This is because the bank-level parallelism exploited by loading multiple sorted streams reduces the row conflicts and enables MeNDA to transpose  $N7$  and  $N8$  faster than  $N6$ .

### 5.6.8 SpMV Analysis

We evaluate SpMV against an HBM-based NMP SpMV accelerator [165]. [165] interleaves the output vector elements among reduction trees to reduce the on-chip buffer to a feasible size, taking advantage of the regular output data. However, sparse matrix transposition outputs an irregular sparse matrix, which has an unknown number of elements per row/column. Therefore, [165] cannot perform sparse matrix transposition without introducing frequent synchronization and large on-chip buffers, which will severely degrade the

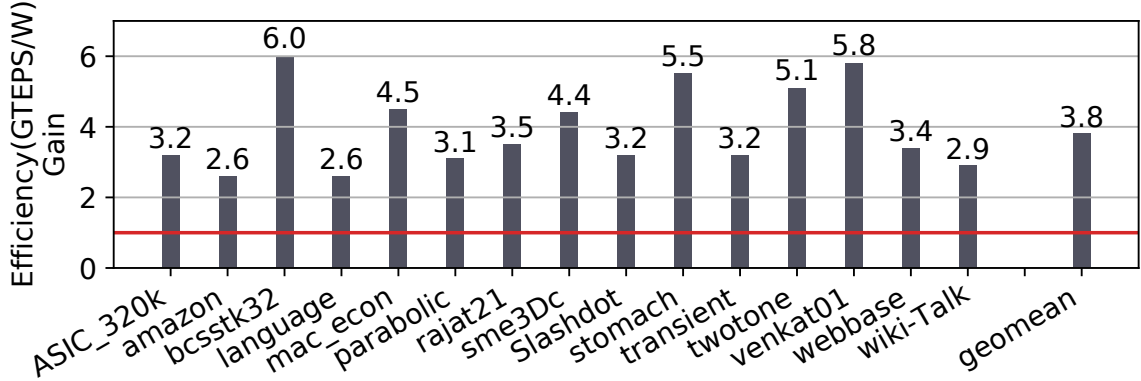


Figure 5.16: Energy efficiency gain of MeNDA over Sadi *et al.* [165] for SpMV.

performance. While [165] is a monolithic design with a high peak throughput saturating the memory bandwidth of four HBM stacks, MeNDA features lightweight PUs that can be integrated into commodity DIMMs, which have better capacity scalability than HBM devices. For a fair comparison, we use Giga Traversed Edges Per Second (GTEPS) per bandwidth (GB/s) as the performance metric. As [165] achieves 0.049 GTEPS/(GB/s) on average, MeNDA achieves a comparable average iso-bandwidth throughput of 0.043 GTEPS/(GB/s) with a maximum of 0.073 GTEPS/(GB/s). For efficiency gain, we scale our power to match the technology while keeping the performance because the performance of MeNDA is limited by the memory bandwidth instead of the system frequency. Overall, MeNDA presents an average improvement of  $3.8\times$  in efficiency (GTEPS/W) (Figure 5.16).

## 5.7 Related Works

**Near-DRAM Accelerators** In recent years, many near-DRAM accelerators have been proposed to accelerate memory bandwidth bound workloads and save data transfer energy. Chameleon integrates CGRAs into the data buffer chip on load-reduced DIMMs [11]. Inspired by Chameleon, TensorDIMM [117] and RecNMP [97] place accelerators in the DRAM buffer devices to optimize sparse embedding operations in recommender systems. The performance benefits of RecNMP are further demonstrated on AxDIMM, an FPGA-based NMP prototyping and evaluation platform [98]. Fafnir identifies the limitations of TensorDIMM and RecNMP and proposes a near-DRAM reduction tree consisting of custom PEs for sparse gathering [8]. GraFboost [92] and MetaStrider [174] are sort-reduce accelerators. GraFboost [92] targets datasets that exceed DRAM capacity and reside in flash-based systems. The intermediate data are reduced by more than 80% before being written back to improve latency and flash lifetime. MetaStrider [174] deploys merger units

and metadata storage at HBM memory controllers and interleaves data by indices at bank level to achieve memory-level parallelism. In sparse matrix transposition, however, there is no data reduction. More importantly, data interleaving can cause output data fragmentation and create difficulties in quickly locating specific NZEs post-merge. In summary, none of the above designs can perform sparse matrix transposition efficiently as is.

There are also designs placing accelerators at bank (group) level to further exploit the inherent parallelism in DRAM devices [50, 103, 32, 104, 102, 122]. However, these designs are mostly used for element-wise or multiply-and-accumulate operations because they require all input operands to sit within a specific bank (group). This is infeasible for sparse matrix transposition as it would pose challenges not only to restricting the required input operands to reside in a bank (group) but also to locating elements in the output matrix after sparse matrix transposition.

**Hybrid Memory Cube (HMC)/HBM accelerators for SpMV and graph analytics** Apart from near-DRAM accelerators, plenty of designs have been proposed to tightly integrate computation logic with 3D/2.5D-stacked memory devices to optimize sparse linear algebra applications, such as SpMV and graph algorithms [165, 200, 5, 144, 37, 213, 218]. These designs usually involve communications between NMP cores, which are prohibitively expensive for near-DRAM accelerators. Besides, HBM/HMC devices often suffer from limited capacity whereas capacity scalability is critical in sparse linear algebra workloads due to the exploding dataset sizes.

None of the beforehand mentioned works address sparse matrix transposition, nor can they be used to perform sparse matrix transposition, including those designs featuring near-memory reduction trees that can compute SpMV [8, 165, 92, 174]. However, based on the insights provided in the prior works, we identify sparse matrix transposition as a promising candidate for NMP because of its low arithmetic complexity and high memory bandwidth requirements.

## 5.8 Conclusion

MeNDA is a scalable solution to near-DRAM multi-way merge for sparse dataflows, including sparse matrix transposition and SpMV. A MeNDA PU features a high-performance merge tree enhanced with techniques to maximize bandwidth utilization. To ease the deployment of MeNDA, a heterogeneous programming model is designed and showcased by integrating MeNDA with a recent graph analytics framework. Overall, MeNDA achieves an average speedup of  $19.1\times$  over `scanTrans` and  $12.0\times$  over `mergeTrans` on CPU and  $7.7\times$  over `cuSPARSE` on GPU for sparse matrix transposition, and shows an average

efficiency gain of  $3.8\times$  over an HBM-based SpMV accelerator. Incurring a power overhead of 78.6 mW per PU, MeNDA can be accommodated by commodity DIMMs, introducing a small area and power overhead.



## CHAPTER 6

### Conclusion

Data in modern applications are not only growing with increasing volume and velocity but also with higher sparsity. The computations on these enormous, sparse data structures pose unique challenges to contemporary computing systems due to the irregular memory access pattern and large memory footprints. With the stagnating scaling of transistor size and power due to the demise of Moore’s law and Dennard scaling, the pressure is on computer architects to develop system designs with better performance and efficiency, giving rise to many emerging architectures. Specialized hardware accelerators are put forward to adopt custom hardware designs that are specifically tuned for the desired dataflows and are free of extraneous logic that does not contribute to the performance of the target workload. On top of that, reconfigurable architectures are developed to trade off some performance and efficiency for better flexibility to address the diversity and fast algorithmic evolution in modern applications. Finally, as memory becomes an increasingly significant bottleneck, processing logic is moved closer or even into the memory subsystems to save the costly data movement. This dissertation proposes to apply these emerging architecture techniques to accelerate sparse linear algebra, starting with a parallelism analysis on contemporary processors to expose the current architecture design trend.

To cover the diverse algorithms and data characteristics in sparse linear algebra, Transmuter is proposed as a programmable and reconfigurable architecture consisting of massively-parallel general-purpose cores connected to a reconfigurable on-chip memory hierarchy. Transmuter achieves an average throughput (energy-efficiency) improvement of  $5.0\times$  ( $18.4\times$ ) and  $4.2\times$  ( $4.0\times$ ) over a high-end CPU and GPU, respectively, across a wide spectrum of commonly used kernels. The true efficacy of Transmuter actually lies in its outstanding performance for sparse linear algebra kernels, which is an energy efficiency within  $4.1\times$  compared to state-of-the-art ASICs for SpMM and SpMV.

Based on the observation that the performance of sparse linear algebra kernels on Transmuter can benefit significantly from carefully chosen algorithms and hardware configurations, CoSPARSE is designed to enhance the performance of Transmuter on SpMV and

graph analytics applications. As an intelligent framework that guides the runtime reconfiguration, CoSPARSE automatically determines the best-performing algorithm and hardware configuration during execution and demonstrates an average speedup and energy efficiency improvement of  $1.5\times$  and  $404.4\times$ , respectively, compared to a recent graph processing framework on a high-end server-class CPU.

The final part of the dissertation proposes MeNDA, a near-memory multi-way merge solution for sparse matrix transposition, which is widely used in sparse linear algebra applications, including the preprocessing stage. MeNDA deploys custom accelerators in the data buffer chips to expose the high internal DRAM memory bandwidth while minimizing hardware modifications to DRAM devices. Because of the wide application of sparse data merging, MeNDA can be easily adapted to support other sparse kernels. Compared to two recently proposed implementations on CPU and a sparse library on GPU, MeNDA achieves a speedup of  $19.1\times$ ,  $12.0\times$ , and  $7.7\times$  respectively, while introducing a minor power overhead of 78.6 mW per DRAM rank. More importantly, MeNDA provides a fast runtime graph transposition implementation to graph analytics frameworks like CoSPARSE so that they no longer need to store more than two copies of the input graph to support runtime dataflow reconfiguration.

While this dissertation covers optimizations for sparse linear algebra execution from hardware architecture designs to software scheduling strategies, there is still scope for future research opportunities. First, because of the flexibility of Transmuter, mapping applications to the Transmuter architecture involves complicated trade-off considerations, posing challenges to library programmers. Creating efficient kernel implementations on Transmuter requires knowledge of both the kernel characteristics and the various Transmuter configurations. To further improve ease of adoption, software tools that can generate efficient kernel implementations on Transmuter or perform fast design space exploration for possible algorithm mappings are desired. Second, Transmuter applies lightweight general-purpose cores as GPEs, which improves programmability by supporting a standard ISA but may result in redundant instructions after compilation as a GNU compiler is used. Further optimizations can be made at the compiler level to tailor the compiled instructions to become most efficient for the Transmuter architecture. Finally, as data sizes keep exploding, it is highly possible for sparse matrices to become too large to fit in the main memory. To accommodate these datasets for near-memory sparse data merging, potential solutions include extending the technique applied in MeNDA to deploy custom PUs in secondary storage, which has different hardware design constraints, or applying data compression techniques to reduce the sizes of the datasets without losing the information.

## BIBLIOGRAPHY

- [1] Seventh green graph 500 list. <http://green.graph500.org/lists.php>, 2016.
- [2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.
- [3] Nilmini Abeyratne, Reetuparna Das, Qingkun Li, Korey Sewell, Bharan Giridhar, Ronald G. Dreslinski, David Blaauw, and Trevor Mudge. Scaling towards kilo-core processors with asymmetric high-radix topologies. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 496–507. IEEE Computer Society, 2013.
- [4] Edward H Adelson, Charles H Anderson, James R Bergen, Peter J Burt, and Joan M Ogden. Pyramid methods in image processing. *RCA Engineer*, 29(6):33–41, 1984.
- [5] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyong Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 105–117, 2015.
- [6] Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, Massoud Pedram, and Muhammad Shafique. X-cgra: An energy-efficient approximate coarse-grained reconfigurable architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.
- [7] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M Kakade, and Matus Telgarsky. Tensor decompositions for learning latent variable models. *Journal of machine learning research*, 15:2773–2832, 2014.
- [8] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung-Kyu Lim, and Hyesoon Kim. Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 908–920, 2021.

- [9] Bahar Asgari, Ramyad Hadidi, and Hyesoon Kim. Ascella: Accelerating sparse computation by enabling stream accesses to memory. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 318–321. IEEE, 2020.
- [10] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. Alrescha: A lightweight reconfigurable sparse-computation accelerator. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 249–260. IEEE, 2020.
- [11] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.
- [12] Tuba Ayhan, Wim Dehaene, and Marian Verhelst. A 128~2048/1536 point fft hardware implementation with output pruning. In *2014 22nd European Signal Processing Conference (EUSIPCO)*, pages 266–270. IEEE, 2014.
- [13] Ariful Azad, Aydin Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811. IEEE, 2015.
- [14] David F Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013.
- [15] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):1–25, 2017.
- [16] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002*, pages 73–78. IEEE, 2002.
- [17] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE, 2012.
- [18] Nathan Bell, Steven Dalton, and Luke N Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing*, 34(4):C123–C152, 2012.
- [19] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Citeseer, 2008.
- [20] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoib Bin Altaf, Nilay

- Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, 2011.
- [21] Nathan L Binkert, Ronald G Dreslinski, Lisa R Hsu, Kevin T Lim, Ali G Saidi, and Steven K Reinhardt. The m5 simulator: Modeling networked systems. *Ieee micro*, 26(4):52–60, 2006.
- [22] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of thread-level parallelism in desktop applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10*, pages 302–313, New York, NY, USA, 2010. ACM.
- [23] Azzedine Boukerche and Carl Tropper. A distributed graph algorithm for the detection of local cycles and knots. *IEEE Transactions on Parallel and Distributed Systems*, 9(8):748–757, 1998.
- [24] Ian Buck. The evolution of gpus for general purpose computing. In *Proceedings of the GPU Technology Conference 2010*, page 11, 2010.
- [25] Aydın Buluç and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.
- [26] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151. IEEE, 2012.
- [27] Sergi Caelles, Kevis-Kokitsi Maninis, Jordi Pont-Tuset, Laura Leal-Taixé, Daniel Cremers, and Luc Van Gool. One-shot video object segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 221–230, 2017.
- [28] Benton Highsmith Calhoun, Joseph F Ryan, Sudhanshu Khanna, Mateja Putic, and John Lach. Flexible circuits and architectures for ultralow power. *Proceedings of the IEEE*, 98(2):267–282, 2010.
- [29] Web Chang. Embedded configurable logic asic, July 10 2001. US Patent 6,260,087.
- [30] Rong Chen, Jiabin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):1–39, 2019.
- [31] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2016.
- [32] Benjamin Y Cho, Jeageun Jung, and Mattan Erez. Accelerating bandwidth-bound deep learning inference with main-memory accelerators. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

- [33] Benjamin Y Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez. Near data acceleration with concurrent host access. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 818–831. IEEE, 2020.
- [34] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.
- [35] Marco Cuturi. Sinkhorn distances: Lightspeed computation of optimal transport. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2, NIPS’13*, pages 2292–2300, 2013.
- [36] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO ’52*, pages 924–939. ACM, 2019.
- [37] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):640–653, 2018.
- [38] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation*, pages 752–768, 2018.
- [39] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rakesh K Gupta, Zhiru Zhang, Ronald G Dreslinski, Christopher Batten, and Michael B Taylor. The celerity open-source 511-core risc-v tiered accelerator fabric: Fast architectures and design methodologies for fast chips. *IEEE Micro*, 38(2):30–41, 2018.
- [40] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.
- [41] Frank Dellaert and Michael Kaess. Square root sam: Simultaneous localization and mapping via square root information smoothing. *The International Journal of Robotics Research*, 25(12):1181–1203, 2006.
- [42] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet’s with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.
- [43] Chris HQ Ding, Xiaofeng He, Hongyuan Zha, Ming Gu, and Horst D Simon. A min-max cut algorithm for graph partitioning and data clustering. In *Proceedings 2001 IEEE International Conference on Data Mining*, pages 107–114. IEEE, 2001.

- [44] Claire Donnat, Marinka Zitnik, David Hallac, and Jure Leskovec. Learning structural node embeddings via diffusion wavelets. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1320–1329. ACM, 2018.
- [45] Richard Dorrance, Fengbo Ren, and Dejan Marković. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 161–170. ACM, 2014.
- [46] Iain S Duff, Michael A Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Transactions on Mathematical Software (TOMS)*, 28(2):239–267, 2002.
- [47] E Swartzlander Earl Jr. Systolic fft processors: Past, present and future. In *IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*, pages 153–158. IEEE, 2006.
- [48] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.
- [49] Nasim Farahini, Shuo Li, Muhammad Adeel Tajammul, Muhammad Ali Shami, Guo Chen, Ahmed Hemani, and Wei Ye. 39.9 gops/watt multi-mode cgra accelerator for a multi-standard basestation. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*, pages 1448–1451. IEEE, 2013.
- [50] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 283–295, 2015.
- [51] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137, 2004.
- [52] Siying Feng, Xin He, Kuan-Yu Chen, Liu Ke, Xuan Zhang, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Menda: A near-memory multi-way merge solution for sparse transposition and dataflows. In *Proceedings of the 49th Annual International Symposium on Computer Architecture, ISCA '22*, page 245–258, New York, NY, USA, 2022. Association for Computing Machinery.
- [53] Siying Feng, Subhankar Pal, Yichen Yang, and Ronald G. Dreslinski. Parallelism analysis of prominent desktop applications: An 18- year perspective. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 202–211, 2019.

- [54] Siying Feng, Jiawen Sun, Subhankar Pal, Xin He, Kuba Kaszyk, Dong-hyeon Park, Magnus Morton, Trevor Mudge, Murray Cole, Michael O’Boyle, Chaitali Chakrabarti, and Ronald Dreslinski. Cosparse: A software and hardware reconfigurable spmv framework for graph analytics. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 949–954, 2021.
- [55] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. Optimizing cuda code by kernel fusion: application on blas. *The Journal of Supercomputing*, 71(10):3934–3957, 2015.
- [56] Kristián Flautner, Rich Uhlig, Steve Reinhardt, and Trevor Mudge. Thread-level parallelism and interactive performance of desktop applications. *SIGOPS Oper. Syst. Rev.*, 34(5):129–138, November 2000.
- [57] Kristián Flautner, Rich Uhlig, Steve Reinhardt, and Trevor Mudge. Thread-level parallelism of desktop applications. *Workshop on Multi-threaded Execution, Architecture and Compilation*, 2000.
- [58] Roger Fletcher. Conjugate gradient methods for indefinite systems. In *Numerical analysis*, pages 73–89. Springer, 1976.
- [59] Roland W Freund and Noël M Nachtigal. Qmr: a quasi-minimal residual method for non-hermitian linear systems. *Numerische mathematik*, 60(1):315–339, 1991.
- [60] Florian Fricke, André Werner, Keyvan Shahin, and Michael Hübner. Cgra tool flow for fast run-time reconfiguration. In *International Symposium on Applied Reconfigurable Computing*, pages 661–672. Springer, 2018.
- [61] Adi Fuchs and David Wentzlaff. The accelerator wall: Limits of chip specialization. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–14. IEEE, 2019.
- [62] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Eda. Data transfer matters for gpu computing. In *2013 International Conference on Parallel and Distributed Systems*, pages 275–282. IEEE, 2013.
- [63] Noriyuki Fujimoto. Dense matrix-vector multiplication on the cuda architecture. *Parallel Processing Letters*, 18(04):511–530, 2008.
- [64] Mingyu Gao and Christos Kozyrakis. Hrl: Efficient and flexible reconfigurable logic for near-data processing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–137. Ieee, 2016.
- [65] Heiner Giefers, Raphael Polig, and Christoph Hagleitner. Measuring and modeling the power consumption of energy-efficient fpga coprocessors for gemm and fft. *Journal of Signal Processing Systems*, 85(3):307–323, December 2016.



- [66] Heiner Giefers, Peter Staar, Costas Bekas, and Christoph Hagleitner. Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of gpu, xeon phi and fpga. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 46–56. IEEE, 2016.
- [67] John R Gilbert, Steve Reinhardt, and Viral B Shah. High-performance graph algorithms from parallel sparse matrices. In *International Workshop on Applied Parallel Computing*, pages 260–269. Springer, 2006.
- [68] John R Gilbert, Steve Reinhardt, and Viral B Shah. A unified framework for numerical and combinatorial computing. *Computing in Science & Engineering*, 10(2):20–25, 2008.
- [69] Andrew Goldberg and Tomasz Radzik. A heuristic improvement of the bellman-ford algorithm. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1993.
- [70] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matthew Moe, and R Reed Taylor. Piplin: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000.
- [71] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.
- [72] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 503–514. IEEE, 2011.
- [73] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao, and Hai Jin. A survey on graph processing accelerators: Challenges and opportunities. *Journal of Computer Science and Technology*, 34(2):339–371, 2019.
- [74] Azzam Haidar, Mark Gates, Stan Tomov, and Jack Dongarra. Toward a scalable multi-gpu eigensolver via compute-intensive kernels and efficient communication. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 223–232. ACM, 2013.
- [75] Tom R Halfhill. Ambric’s new parallel processor. *Microprocessor Report*, 20(10):19–26, 2006.
- [76] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

- [77] Václav Hapla, David Horák, and Michal Merta. Use of direct solvers in tfeti massively parallel implementation. In *International Workshop on Applied Parallel Computing*, pages 192–205. Springer, 2012.
- [78] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.
- [79] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. Sparse-tpu: Adapting systolic arrays for sparse matrices. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–12, 2020.
- [80] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–333, 2019.
- [81] Mark Horowitz. 1.1 computing’s energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14. IEEE, 2014.
- [82] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. Cosa: Scheduling by constrained optimization for spatial accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 554–566. IEEE, 2021.
- [83] Randall A Hughes and John D Shott. The future of automation for high-volume wafer fabrication and asic manufacturing. *Proceedings of the IEEE*, 74(12):1775–1793, 1986.
- [84] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture, ISCA ’07*, pages 186–197. ACM, 2007.
- [85] Satoshi Itoh, Pablo Ordejón, and Richard M Martin. Order-n tight-binding molecular dynamics on parallel computers. *Computer physics communications*, 88(2-3):173–185, 1995.
- [86] Preston A. Jackson, Cy P. Chan, Jonathan E. Scalera, Charles M. Rader, and M. Michael Vai. A systolic fft architecture for real time fpga systems. In *High Performance Embedded Computing Conference (HPEC)*, 2004.

- [87] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11—seamless operability between c++ 11 and python, 2017.
- [88] Supreet Jeloka, Reetuparna Das, Ronald G Dreslinski, Trevor Mudge, and David Blaauw. Hi-rise: a high-radix switch for 3d integration with single-cycle arbitration. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 471–483. IEEE, 2014.
- [89] Kurtis T. Johnson, Ali R Hurson, and Behrooz Shirazi. General-purpose systolic arrays. *Computer*, 26(11):20–31, 1993.
- [90] Rodney W Johnson, Chua-Huang Huang, and John R Johnson. Multilinear algebra and parallel programming. *The Journal of Supercomputing*, 5(2):189–217, 1991.
- [91] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12. Association for Computing Machinery, 2017.
- [92] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Grafboost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 411–424. IEEE Press, 2018.
- [93] Marcin Junczys-Dowmunt, Roman Grundkiewicz, Tomasz Dwojak, Hieu Hoang, Kenneth Heafield, Tom Neckermann, Frank Seide, Ulrich Germann, Alham Fikri Aji, Nikolay Bogoychev, André F. T. Martins, and Alexandra Birch. Marian: Fast neural machine translation in C++. In *Proceedings of ACL 2018, System Demonstrations*, pages 116–121, Melbourne, Australia, July 2018. Association for Computational Linguistics.
- [94] Haim Kaplan, Micha Sharir, and Elad Verbin. Colored intersection searching via sparse rectangular matrix multiplication. In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 52–60, 2006.

- [95] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. Hy-cube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.
- [96] George Karypis, Anshul Gupta, and Vipin Kumar. A parallel formulation of interior point algorithms. In *Supercomputing'94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pages 204–213. IEEE, 1994.
- [97] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 790–803, 2020.
- [98] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, Yeongon Cho, Jin Hyun Kim, Yongsuk Kwon, Kyungsoo Kim, Jin Jung, Ilkwon Yun, Sung Joo Park, Hyunsun Park, Joonho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin Sean Lee. Near-memory processing in action: Accelerating personalized recommendation with axdim. *IEEE Micro*, pages 1–1, 2021.
- [99] John Kelm, Daniel Johnson, Matthew Johnson, Neal Crago, William Tuohy, Aqeel Mahesri, Steven Lumetta, Matthew Frank, and Sanjay Patel. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 140–151. ACM, 2009.
- [100] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy G. Mattson, and José E. Moreira. Mathematical foundations of the graphblas. In *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*, pages 1–9. IEEE, 2016.
- [101] Khubaib, M. Aater Suleman, Milad Hashemi, Chris Wilkerson, and Yale N. Patt. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 305–316, 2012.
- [102] Byeongho Kim, Jongwook Chung, Eojin Lee, Wonkyung Jung, Sunjung Lee, Jaewan Choi, Jaehyun Park, Minbok Wi, Sukhan Lee, and Jung Ho Ahn. Mvid: Sparse matrix-vector multiplication in mobile dram for accelerating recurrent neural networks. *IEEE Transactions on Computers*, 69(7):955–967, 2020.
- [103] Byeongho Kim, Jaehyun Park, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. Trim: Tensor reduction in memory. *IEEE Computer Architecture Letters*, 20(1):5–8, 2021.

- [104] Heesu Kim, Hanmin Park, Taehyun Kim, Kwanheum Cho, Eojin Lee, Soojung Ryu, Hyuk-Jae Lee, Kiyoungh Choi, and Jinho Lee. Gradpim: A practical processing-in-dram architecture for gradient descent. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 249–262, 2021.
- [105] Martha Mercaldi Kim, John D. Davis, Mark Oskin, and Todd Austin. Polymorphic on-chip networks. In *Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA '08*, pages 101–112. IEEE Computer Society, 2008.
- [106] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer architecture letters*, 15(1):45–49, 2015.
- [107] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszal, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018*, pages 296–311. ACM, 2018.
- [108] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. Stash: Have your scratchpad and cache it too. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture, ISCA '15*, pages 707–719. ACM, 2015.
- [109] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [110] Hsiang-Tsung Kung. Why systolic architectures? *IEEE computer*, 15(1):37–46, 1982.
- [111] HT Kung, Bradley McDanel, and Sai Qian Zhang. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 821–834. ACM, 2019.
- [112] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, 2007.
- [113] Ian Kuon, Russell Tessier, and Jonathan Rose. *FPGA architecture: Survey and challenges*. Now Publishers Inc, 2008.
- [114] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger. From word embeddings to document distances. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37, ICML'15*, pages 957–966, 2015.

- [115] Georgi Kuzmanov and Mottaqiallah Taouil. Reconfigurable sparse/dense matrix-vector multiplier. In *2009 International Conference on Field-Programmable Technology*, pages 483–488. IEEE, 2009.
- [116] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.
- [117] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '52*, page 740–753, New York, NY, USA, 2019. Association for Computing Machinery.
- [118] Benjamin C Lee, Richard W Vuduc, James W Demmel, and Katherine A Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *International Conference on Parallel Processing, 2004. ICPP 2004.*, pages 169–176. IEEE, 2004.
- [119] Chang-Chi Lee, CP Hung, Calvin Cheung, Ping-Feng Yang, Chin-Li Kao, Dao-Long Chen, Meng-Kai Shih, Chien-Lin Chang Chien, Yu-Hsiang Hsiao, Li-Chieh Chen, Michael Su, Michael Alfano, Joe Siegel, Julius Din, and Bryan Black. An overview of the development of a gpu with integrated hbm on silicon interposer. In *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*, pages 1439–1444. IEEE, 2016.
- [120] Chang-Hwan Lee. A gradient approach for value weighted classification learning in naive bayes. *Knowledge-Based Systems*, 85:71–79, 2015.
- [121] Dongwook Lee, Manhwee Jo, Kyuseung Han, and Kiyoun Choi. Flora: Coarse-grained reconfigurable architecture with floating-point operation capability. In *2009 International Conference on Field-Programmable Technology*, pages 376–379. IEEE, 2009.
- [122] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyoungwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. Hardware architecture and software stack for pim based on commercial dram technology : Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 43–56, 2021.
- [123] John J Leonard, Hugh F Durrant-Whyte, and Ingemar J Cox. Dynamic map building for an autonomous mobile robot. *The International Journal of Robotics Research*, 11(4):286–298, 1992.
- [124] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1–20, 2016.

- [125] Jiajia Li, Xingjian Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. An optimized large-scale hybrid dgemm design for cpus and ati gpus. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 377–386. ACM, 2012.
- [126] Cao Liang and Xinming Huang. Smartcell: A power-efficient reconfigurable architecture for data streaming applications. In *2008 IEEE Workshop on Signal Processing Systems*, pages 257–262. IEEE, 2008.
- [127] Yuan Lin, Hyunseok Lee, Mark Woh, Yoav Harel, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. Soda: A low-power architecture for software radio. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 89–101. IEEE Computer Society, 2006.
- [128] Leibo Liu, Dong Wang, Min Zhu, Yansheng Wang, Shouyi Yin, Peng Cao, Jun Yang, and Shaojun Wei. An energy-efficient coarse-grained reconfigurable processing unit for multiple-standard video decoding. *IEEE Transactions on Multimedia*, 17(10):1706–1720, 2015.
- [129] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Computing Surveys (CSUR)*, 52(6):1–39, 2019.
- [130] Beth Logan. Mel frequency cepstral coefficients for music modeling. In *ISMIR*, volume 270, pages 1–11, 2000.
- [131] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.
- [132] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.
- [133] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M Sleiman, Ronald Dreslinski, Thomas F Wenisch, and Scott Mahlke. Composite cores: Pushing heterogeneity into a core. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 317–328. IEEE, 2012.
- [134] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. Asic clouds: Specializing the datacenter. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 178–190. IEEE, 2016.
- [135] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 161–171, New York, NY, USA, 2000. ACM.

- [136] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 631–643, 2017.
- [137] Tim Mattson, David A. Bader, Jonathan W. Berry, Aydin Buluç, Jack J. Dongarra, Christos Faloutsos, John Feo, John R. Gilbert, Joseph Gonzalez, Bruce Hendrickson, Jeremy Kepner, Charles E. Leiserson, Andrew Lumsdaine, David A. Padua, Stephen Poole, Steven P. Reinhardt, Mike Stonebraker, Steve Wallach, and Andrew Yoo. Standards for graph algorithm primitives. In *IEEE High Performance Extreme Computing Conference, HPEC 2013, Waltham, MA, USA, September 10-12, 2013*, pages 1–2. IEEE, 2013.
- [138] Patrick J Meaney, Lawrence D Curley, Glenn D Gilda, Mark R Hodges, Daniel J Buerkle, Robert D Siegl, and Roger K Dong. The ibm z13 memory subsystem for big data. *IBM Journal of Research and Development*, 59(4/5):4–1, 2015.
- [139] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. An analysis of neural language modeling at multiple scales. *arXiv preprint arXiv:1803.08240*, 2018.
- [140] Badri Narayan Mohapatra and Rashmita Kumari Mohapatra. Fft and sparse fft techniques and applications. In *2017 Fourteenth International Conference on Wireless and Optical Communications Networks (WOCN)*, pages 1–5. IEEE, 2017.
- [141] Frank Mueller. Pthreads library interface. *Florida State University*, 1993.
- [142] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27:28, 2009.
- [143] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A modern primer on processing in memory, 2020.
- [144] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International symposium on high performance computer architecture (HPCA)*, pages 457–468. IEEE, 2017.
- [145] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 456–471, 2013.
- [146] Chris Nicol. A coarse grain reconfigurable array (cgra) for statically scheduled data flow computing. *Wave Computing White Paper*, 2017.
- [147] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA '17*, pages 416–429. ACM, 2017.



- [148] Molly A O’Neil and Martin Burtscher. Microarchitectural performance characterization of irregular gpu kernels. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 130–139. IEEE, 2014.
- [149] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2(11):1–4, 2015.
- [150] Mike O’Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W Keckler, and William J Dally. Fine-grained dram: energy-efficient dram for extreme bandwidth systems. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 41–54. IEEE, 2017.
- [151] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736, Feb 2018.
- [152] Subhankar Pal, Siying Feng, Dong-hyeon Park, Sung Kim, Aporva Amarnath, Chi-Sheng Yang, Xin He, Jonathan Beaumont, Kyle May, Yan Xiong, Kuba Kaszyk, John Magnus Morton, Jiawen Sun, Michael O’Boyle, Murray Cole, Chaitali Chakrabarti, David Blaauw, Hun-Seok Kim, Trevor Mudge, and Ronald Dreslinski. Transmuter: Bridging the efficiency gap using memory and dataflow reconfiguration. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 175–190, 2020.
- [153] Subhankar Pal, Dong-Hyeon Park, Siying Feng, Paul Gao, Jielun Tan, Austin Rovinski, Shaolin Xie, Chun Zhao, Aporva Amarnath, Timothy Wesley, Jonathan Beaumont, Kuan-Yu Chen, Chaitali Chakrabarti, Michael Bedford Taylor, Trevor N. Mudge, David T. Blaauw, Hun-Seok Kim, and Ronald G. Dreslinski. A 7.3 M output non-zeros/j sparse matrix-matrix multiplication accelerator using memory reconfiguration in 40 nm. In *2019 Symposium on VLSI Circuits, Kyoto, Japan, June 9-14, 2019*, page 150. IEEE, 2019.
- [154] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315. IEEE, 2019.
- [155] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

- [156] Dong-Hyeon Park, Subhankar Pal, Siying Feng, Paul Gao, Jielun Tan, Austin Rovinski, Shaolin Xie, Chun Zhao, Aporva Amarnath, Timothy Wesley, Jonathan Beaumont, Kuan-Yu Chen, Chaitali Chakrabarti, Michael Bedford Taylor, Trevor N. Mudge, David T. Blaauw, Hun-Seok Kim, and Ronald G. Dreslinski. A 7.3 M output non-zeros/j, 11.7 M output non-zeros/gb reconfigurable sparse matrix-matrix multiplication accelerator. *Journal of Solid-State Circuits*, 55(4):933–944, 2020.
- [157] Ardavan Pedram, Andreas Gerstlauer, and Robert A Van De Geijn. A high-performance, low-power linear algebra core. In *ASAP 2011-22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 35–42. IEEE, 2011.
- [158] Ardavan Pedram, John D. McCalpin, and Andreas Gerstlauer. A highly efficient multicore floating-point fft architecture based on hybrid linear algebra/fft cores. *Journal of Signal Processing Systems*, 77(1):169–190, Oct 2014.
- [159] Gerald Penn. Efficient transitive closure of sparse matrices over closed semirings. *Theoretical Computer Science*, 354(1):72–81, 2006.
- [160] C.A. Philips. Parallel graph contraction. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 148–157, 1989.
- [161] Kara KW Poon, Steven JE Wilton, and Andy Yan. A detailed power model for field-programmable gate arrays. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 10(2):279–302, 2005.
- [162] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402. IEEE, 2017.
- [163] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, June 2014.
- [164] Karl Rupp. 42 years of microprocessor trend data. <https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>. Accessed: 2019-10-01.
- [165] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C Hoe, Larry Pileggi, and Franz Franchetti. Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 347–358, 2019.

- [166] Tim Salimans, Han Zhang, Alec Radford, and Dimitris Metaxas. Improving gans using optimal transport. *arXiv preprint arXiv:1803.05573*, 2018.
- [167] Fabian Schuiki, Michael Schaffner, and Luca Benini. Ntx: An energy-efficient streaming accelerator for floating-point generalized reduction workloads in 22 nm fd-soi. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 662–667. IEEE, 2019.
- [168] Korey Sewell, Ronald G. Dreslinski, Thomas Manville, Sudhir Satpathy, Nathaniel Ross Pinckney, Geoffrey Blake, Michael Cieslak, Reetuparna Das, Thomas F. Wenisch, Dennis Sylvester, David T. Blaauw, and Trevor N. Mudge. Swizzle-switch networks for many-core systems. *IEEE J. Emerg. Sel. Topics Circuits Syst.*, 2(2):278–294, 2012.
- [169] Muhammad Shafique and Siddharth Garg. Computing in the dark silicon era: Current trends and research challenges. *IEEE Design & Test*, 34(2):8–23, 2016.
- [170] Viral B Shah. *An interactive system for combinatorial scientific computing with an emphasis on programmer productivity*. PhD thesis, 2007.
- [171] Julian Shun. Ligra: A lightweight graph processing framework for shared memory. url= <http://jshun.github.io/ligra/>, 2013.
- [172] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.
- [173] Anuraag Soorishetty, Jian Zhou, Subhankar Pal, David Blaauw, H Kim, Trevor Mudge, Ronald Dreslinski, and Chaitali Chakrabarti. Accelerating linear algebra kernels on a massively parallel reconfigurable architecture. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1558–1562. IEEE, 2020.
- [174] Sriseshan Srikanth, Anirudh Jain, Joseph M. Lennon, Thomas M. Conte, Erik Debenedictis, and Jeanine Cook. Metastrider: Architectures for scalable memory-centric reduction of sparse data streams. *ACM Trans. Archit. Code Optim.*, 16(4), oct 2019.
- [175] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonese, and Zhiru Zhang. Matraport: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 766–780. IEEE, 2020.
- [176] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonese, and Zhiru Zhang. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 689–702. IEEE, 2020.

- [177] Samuel Steffl and Sherief Reda. Lacore: A supercomputing-like linear algebra accelerator for soc-based designs. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 137–144. IEEE, 2017.
- [178] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a functional data-parallel ir for high-performance gpu code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 74–85. IEEE, 2017.
- [179] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.
- [180] Weiyi Sun, Zhaoshi Li, Shouyi Yin, Shaojun Wei, and Leibo Liu. Abc-dimm: Alleviating the bottleneck of communication in dimm-based near-memory processing with inter-dimm broadcast. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 237–250. IEEE, 2021.
- [181] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015.
- [182] Cheng Tan, Manupa Karunaratne, Tulika Mitra, and Li-Shiuan Peh. Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 575–587. IEEE, 2018.
- [183] Masakazu Tanomoto, Shinya Takamaeda-Yamazaki, Jun Yao, and Yasuhiko Nakashima. A cgra-based approach for accelerating convolutional neural networks. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pages 73–80. IEEE, 2015.
- [184] Michael Bedford Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *DAC Design Automation Conference 2012*, pages 1131–1136, June 2012.
- [185] Michael Bedford Taylor, Jason Sungtae Kim, Jason E. Miller, David Wentzlaff, Fae Ghodrati, Ben Greenwald, Henry Hoffmann, Paul R. Johnson, Jae W. Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpfen, Matthew I. Frank, Saman P. Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.
- [186] A Tech. Nvidia launches tesla k40. <http://www.anandtech.com/show/7521/nvidia-launches-tesla-k40>, 2013.

- [187] Vaishali Tehre, Pankaj Agrawal, and RV Kshrisagar. Implementation of fast fourier transform accelerator on coarse grain reconfigurable architecture. *International Journal of Computer Science and Network (IJCSN)*, pages 955–959, 2016.
- [188] Stijn Marinus Van Dongen. *Graph clustering by flow simulation*. PhD thesis, 2000.
- [189] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. Scaleddeep: A scalable compute architecture for learning and evaluating deep networks. *ACM SIGARCH Computer Architecture News*, 45(2):13–26, 2017.
- [190] Manish Verma, Lars Wehmeyer, Peter Marwedel, and Peter Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2, DATE '04*, pages 21264–. IEEE Computer Society, 2004.
- [191] Kizheppatt Vipin and Suhaib A Fahmy. Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys (CSUR)*, 51(4):1–39, 2018.
- [192] James Vlasblom and Shoshana J Wodak. Markov clustering versus affinity propagation for the partitioning of protein interaction graphs. *BMC bioinformatics*, 10(1):1–14, 2009.
- [193] Donglin Wang, Xueliang Du, Leizu Yin, Chen Lin, Hong Ma, Weili Ren, Huijuan Wang, Xingang Wang, Shaolin Xie, Lei Wang, Zijun Liu, Tao Wang, Zhonghua Pu, Guangxin Ding, Mengchen Zhu, Lipeng Yang, Ruoshan Guo, Zhiwei Zhang, Xiao Lin, Jie Hao, Yongyong Yang, Wenqin Sun, Fabiao Zhou, NuoZhou Xiao, Qian Cui, and Xiaoqin Wang. Mapu: A novel mathematical computing architecture. *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 457–468, 2016.
- [194] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on gpu. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 38–52, 2019.
- [195] Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-chun Feng. Parallel transposition of sparse data structures. In *Proceedings of the 2016 International Conference on Supercomputing, ICS '16*, New York, NY, USA, 2016. Association for Computing Machinery.
- [196] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 1–12, 2016.

- [197] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. Enabling fpgas in hyperscale data centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pages 1078–1086. IEEE, 2015.
- [198] Mark Wijtvliet, Luc Waeijen, and Henk Corporaal. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 235–244. IEEE, 2016.
- [199] Samuel Williams. Roofline: An insightful visual performance model for floating-point programs and multicore. *ACM Communications*, 2009.
- [200] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. Spacea: Sparse matrix vector multiplication on processing-in-memory accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 570–583. IEEE, 2021.
- [201] Xilinx. *Partial Reconfiguration User Guide UG702 (v13.3)*.
- [202] Xilinx. *Partial Reconfiguration User Guide UG909 (v2018.1)*.
- [203] Yan Xiong, Jian Zhou, Subhankar Pal, David Blaauw, Hun-Seok Kim, Trevor Mudge, Ronald Dreslinski, and Chaitali Chakrabarti. Accelerating deep neural network computation on a low power reconfigurable architecture. In *2020 International Symposium on Circuits and Systems (ISCAS)*, page to appear. IEEE, 2020.
- [204] Jinchao Xu and Ludmil Zikatanov. Algebraic multigrid methods. *Acta Numerica*, 26:591–721, 2017.
- [205] Ichitaro Yamazaki and Xiaoye S Li. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *International Conference on High Performance Computing for Computational Science*, pages 421–434. Springer, 2010.
- [206] Carl Yang, Aydın Buluç, and John D Owens. Implementing push-pull efficiently in graphblas. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–11, 2018.
- [207] Fanghua Ye, Chuan Chen, and Zibin Zheng. Deep autoencoder-like nonnegative matrix factorization for community detection. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 1393–1402, 2018.
- [208] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. Speeding up spmv for power-law graph analytics by enhancing locality & vectorization. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

- [209] Raphael Yuster and Uri Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '04*, page 254–260, USA, 2004. Society for Industrial and Applied Mathematics.
- [210] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.
- [211] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. Gamma: leveraging gustavson’s algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 687–701, 2021.
- [212] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 183–193, 2015.
- [213] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. Graphp: Reducing communication for pim-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 544–557. IEEE, 2018.
- [214] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–274. IEEE, 2020.
- [215] Jinhong Zhou, Shaoli Liu, Qi Guo, Xuda Zhou, Tian Zhi, Daofu Liu, Chao Wang, Xuehai Zhou, Yunji Chen, and Tianshi Chen. Tunao: A high-performance and energy-efficient reconfigurable accelerator for graph processing. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CC-GRID)*, pages 731–734. IEEE, 2017.
- [216] Qiuling Zhu, Tobias Graf, H. Ekin Sumbul, Lawrence T. Pileggi, and Franz Franchetti. Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware. *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2013.
- [217] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 301–316, 2016.
- [218] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. Graphq: Scalable pim-based graph processing. In *Proceedings of*

*the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 712–725, 2019.