Optimizing Sparse Linear Algebra on Reconfigurable Architecture

by

Dong-hyeon Park

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engieering)
in the University of Michigan
2021

Doctoral Committee:

Professor Trevor Mudge, Chair
Professor David Blaauw
Assistant Professor Ronald Dreslinski
Assistant Professor Hun Seok Kim

Dong-hyeon Park

dohypark@umich.edu

ORCID iD: 0000-0003-3052-6890

DEDICATION

I dedicate this dissertation to my parents, Hyun-Gyu Park and Myeongsook Hyun, for all the love, dedication, and sacrifice they have given me to provide me with the best environment to pursue my dreams. They were always there to support me at every step of my life.

ACKNOWLEDGMENTS

My greatest thank goes to my advisor, Professor Trevor Mudge, for all the support, wisdom, and patience he has given me throughout my time at Michigan. I would not have been able to get all the way here without his help and guidance. I would also like to thank my dissertation committee members, Professor David Blaauw, Ron Dreslinski, and Hun-Seok Kim, for their insights and guidance through the various DARPA projects I had with them, as well as the final steps of my studies.

All the works in this dissertation were possible thanks to the great colleagues and friends I met throughout my years at Michigan. I owe many of my works to Subhankar Pal, who led the Transmuter project and was instrumental in providing me with a platform to test all the various ideas I came up with throughout the years. My other close collaborators, Siying Feng, Xin He, Aprova Amarnath, Jonathan Beaumont, Haojie Ye, Yuhan Chen, Sung Kim, and Jielun Tan, were all instrumental to my studies and I will cherish the memories I shared with them.

Last but not least, none of my accomplishments would have been possible without all the love and care of my parents and my sister, Dong-min Park. It is all thanks to them that I was able to get here and be who I am.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

ABSTRACT

The rise of cloud computing and deep machine learning in recent years have led to a tremendous growth of workloads that are not only large, but also have highly sparse representations. A large fraction of machine learning problems are formulated as sparse linear algebra problems in which the entries in the matrices are mostly zeros. Not surprisingly, optimizing linear algebra algorithms to take advantage of this sparseness is critical for efficient computation on these large datasets.

This thesis presents a detailed analysis of several approaches to sparse matrix-matrix multiplication, a core computation of linear algebra kernels. While the arithmetic count of operations for the nonzero elements of the matrices are the same regardless of the algorithm used to perform matrix-matrix multiplication, there is significant variation in the overhead of navigating the sparse data structures to match the nonzero elements with the correct indices. This work explores approaches to minimize these overheads as well as the number of memory accesses for sparse matrices. To provide concrete numbers, the thesis examines inner product, outer product and row-wise algorithms on Transmuter, a flexible accelerator that can reconfigure its cache and crossbars at runtime to meet the demands of the program. This thesis shows how the reconfigurability of Transmuter can improve complex workloads that contain multiple kernels with varying compute and memory requirements, such as the Graphsage deep neural network and the Sinkhorn algorithm for optimal transport distance. Finally, we examine a novel Transmuter feature: register-to-register queues for efficient communication between its processing elements, enabling systolic array style computation for signal processing algorithms.

CHAPTER 1

Introduction

1.1 Motivation

The emergence of big data and massive social networks, as well as growing interest in artificial intelligence have resulted in greater interest graph and matrix-based workloads with direct real-world applications [65, 79, 17]. Speeding up these complex workloads require not only efficient software, but fast hardware that can meet the emerging demands and constraints of the workloads. However, due to the fall of Moore's Law, one can no longer rely on process technology for significant improvement in processor performance. As shown in Figure 1.1, the saturation of single-thread processing and limitations in power consumption have made it difficult to improve performance through the traditional, homogeneous computing model. Thus, to extract more power from limited number of available transistors, computer architects have gravitated towards more specialized, application-specific designs.

From GPUs, to FPGAs and ASICs, there exists several different platforms for accelerating specialized workloads, each with their own unique benefits and challenges. The rapidly evolving landscape of big data and artificial intelligence workloads have made adaptability an important quality for future architectures. An application-specific hardware that is limited to a narrow pool of algorithms cannot meet the demands of any new applications that emerges to replace the old. Not only that, due to the wide range of different kernels and workloads that are relevant in today's computing environment, the optimal hardware for a certain algorithm can change depending on

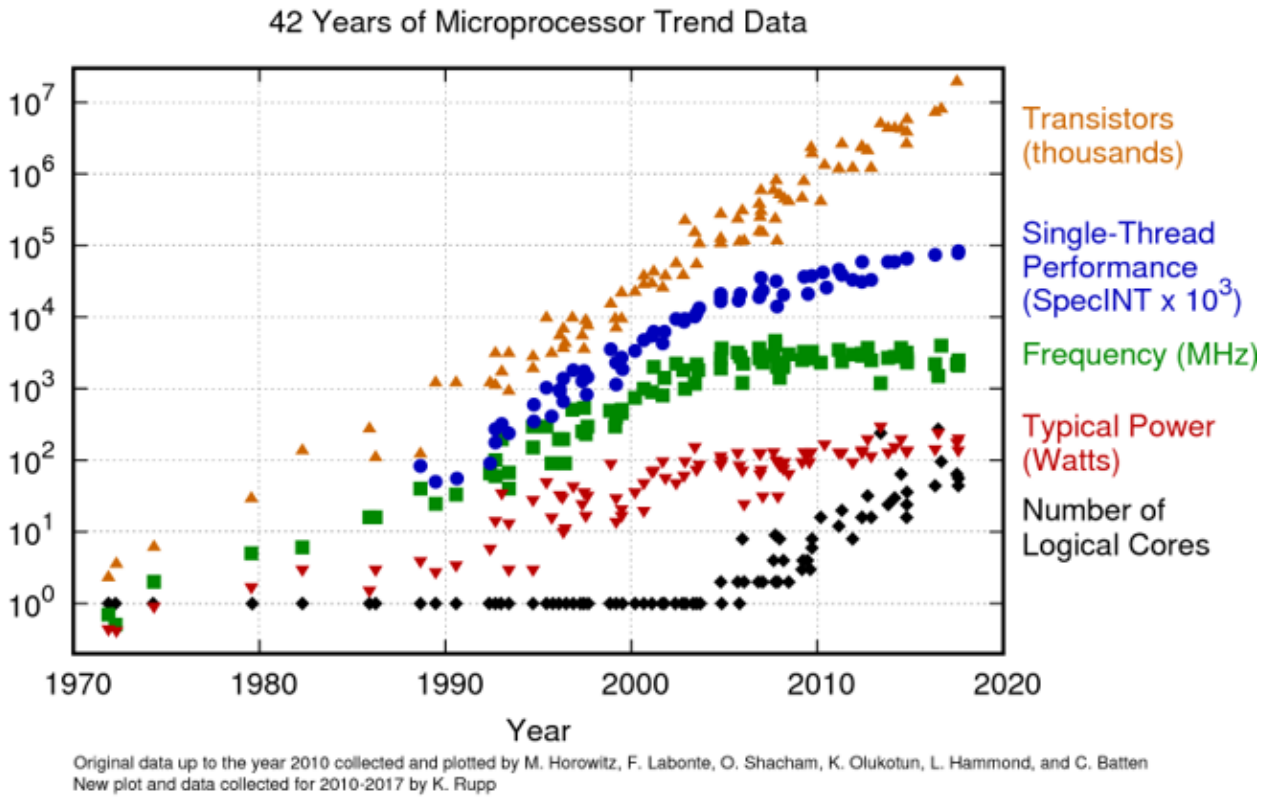


Figure 1.1: Trend in transistor and microprocessor performance over the past 42 years. [59]

the nature of the data the algorithm is processing. Therefore, while architects need to depend on customized hardware to accelerate key kernels, there must be enough flexibility to encompass a wide range of different behaviors. A great way to achieve this balance is the reconfigurable architecture.

This thesis presents Transmuter, a reconfigurable architecture that can reconfigure its cache and crossbars on-the-fly was developed to satisfy the varying demands of sparse workloads. Various different approaches to sparse linear algebra are explored, with an emphasis on outer-product and rowwise sparse matrix-matrix multiplication. In addition, the benefits of reconfiguration in other emerging workloads, such as as fast-fourier transform, Graphsage deep neural network, Sinkhorn algorithm for optimal transport distance,, and smith-waterman algorithm for genomic sequencing were studied in detail.

The overall design and composition of the Transmuter is presented in Chapter 2, along with the OuterSpace chip that served as a prototype design for the Transmuter. Sparse linear algebra is

explored in detail in Chapter 3, with a heavy emphasis on sparse matrix-matrix multiplication. The adaptations made on fast-fourier transform, Graphsage, and Sinkhorn workloads for Transmuter platform are presented in Chapters 4, 5, and 6 respectively. Finally, the benefit of a scalable vector unit for genomic sequencing is studied in Chapter 7.

1.2 Background

1.2.1 Reconfigurable Architecture

An early reconfigurable computer by Mai *et al.* [46] proposes a modular architecture that can be configured to mimic two distinct machines: a VLIW system with vast data parallelism and a speculative multiprocessor supporting irregular workloads. Another early proposal for reconfigurable computers is the Raw Microprocessor [71]. It implements a tiled architecture focusing on scalability in the face of increasing wire delays. This parallel architecture exposes a familiar programming interface, supporting traditional multi-threaded code. The architecture focuses on developing an efficient, distributed interconnect and integrating it tightly into the ISA.

There have been a few proposals that reconfigure at the core level. Core Fusion allows cores in a CMP machine to be "fused" into a large powerful CPU or "split" into many small independent cores [29]. Govindaraju *et al.* [24] integrate a reconfigurable fabric into a programmable core as a set of functional units, mapping performance-critical operations on to the fabric. While this architecture allows the frequent computations to be accelerated, it does not focus on accelerating memory-bound workloads. MorphCore modifies an out-of-order core to also support a highly-threaded in-order SMT core [34].

The network interconnect is another area of reconfiguration. Kim et al. [35] proposes a polymorphic on-chip network that can be customized for each application prior to runtime. Recently, Stitch [70] introduced a many-core architecture consisting of heterogeneous accelerators and integrated configurable instruction set extensions to an existing ISA to reduce the overhead of adding those compute resources to the microarchitecture. The architecture allows the reconfigurable units

to be shared between cores, increasing the available compute resources without adding the overhead for each core.

1.2.2 Programmability

Many recent works have focused on specialized hardware, but with limited semantics that restrict the user to program for specific architectures.

Gao *et al.* [20] introduced heterogeneous reconfigurable logic (HRL) as a substrate for near-data processing. HRL exploits the high bandwidth of 3D stacked memories by combining coarse- and fine-grained reconfigurable blocks into a compute fabric. However, it needs to be programmed using a hardware description language (HDL). Many efforts have been made to abstract away the hardware details from programmers. Rigel [32] is a tiled many-core accelerator that uses a custom programming model, *Rigel task model*, to efficiently manage 1,024 cores through task-based software APIs. Wang *et al.* proposes MaPU [74], a mathematical architecture which features a multi-granularity parallel memory system that enables concurrent accesses to configurable byte-widths as well as a state-machine-based program model.

In a recent work, Nowatzki *et al.* assert that current programming models express algorithms inefficiently and propose a new programming model, Stream Dataflow [51]. The programming model requires that the algorithm be expressed as a data flow graph, with the inputs and outputs specified as streams with a particular access pattern.

Plasticine [58] is a reconfigurable accelerator for parallel patterns. The authors propose a network of Pattern Compute Units (PCUs) and Pattern Memory Units (PMUs) that can be configured at compile-time to suit the needs of particular algorithm, but the program has to be expressed in a Domain-Specific Language (DSL). While programming models and DSLs are an improvement over the lower-level HDLs, the barrier-to-entry is still higher than conventional CPUs and GPUs.

1.2.3 Sparse Matrix Multiplication

One of the fundamental kernels that drives graph analytic and machine learning workloads is matrix multiplication. Traditionally, matrix multiplication workloads focused on performing linear algebra operations on dense matrices that depended primarily on high compute throughput. However, modern datacenter workloads are large and extremely sparse, where a majority of their contents are zeros. Thus, there has been an increase in attention towards matrix multiplication algorithms that target large sparse matrices, such as the adjacency matrix of Facebook friendships which is of size 1.08 B$\times$1.08 B but with only 0.0003% Non-Zero Elements (NZEs) [52].

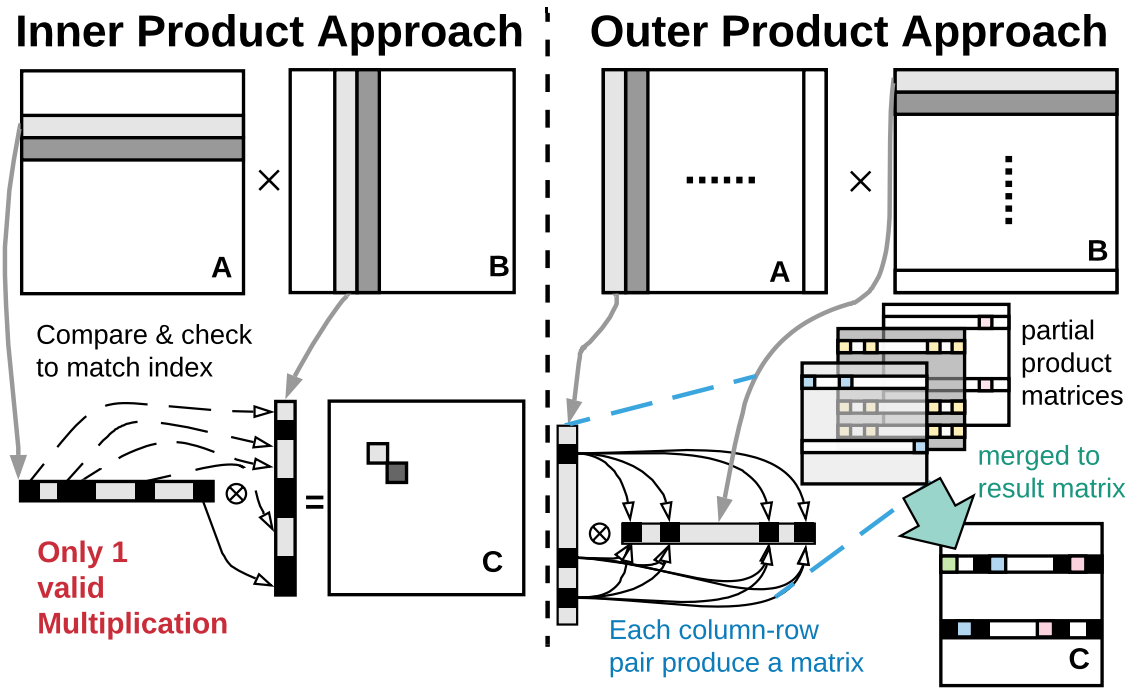


Figure 1.2: High-level overview of sparse matrix-matrix multiplication using inner product and outer product methods.

Since the number of zero elements in a sparse matrix largely outnumber the Number of Non-Zeros (NNZs), it is prudent to store them in *compressed formats*. The Compressed Sparse Row (CSR) format is a standard for storing sparse matrices in graph analytics, scientific computation,

etc [60]. It represents an $N \times N$ sparse matrix using three arrays – *values*, *column-indices* and *row-pointers*, with a total storage overhead of $2 \cdot \text{NNZ} + N + 1$. Compressed Sparse Column (CSC) is the transposed form of CSR, where the latter two arrays are replaced by *row-indices* and *column-pointers*, respectively.

Standard inner-product based multiplication algorithms are not suitable for processing extremely large, sparse matrices, because the majority of computations are wasted on processing zero-value elements. Efficient sparse matrix-matrix multiplication (SpMM) that focuses only on the NZEs greatly improve the performance of such workloads. The SpMM kernel is memory-bound rather than compute-bound, due to poor data locality and compute-to-communication ratio. Thus, accelerating SpMM requires eliminating redundant memory accesses and maximizing data reuse.

The most common implementation of matrix-matrix multiplication is the inner product method, as shown in Figure 1.2. In the inner product method, a row of the first operand is multiplied by the column of the second operand to produce a single element in the result matrix. While this approach works efficiently for dense matrices, once the matrices become too sparse, a significant portion of the runtime is spent on index matching the two operands to find nonzero elements with the same row or column indices. This results in low NNZs per byte fetched from off-chip, leading to *unproductive loads*. Limited on-chip storage further forces repetitive fetching of the same data, worsening the memory bottleneck.

To eliminate the wasted index matching and ensure all memory loads are productive, we employ an outer product algorithm that we first proposed in [52]. Unlike the inner product approach, the outer product approach multiplies the columns of the first operand with the rows of the second operand to generate partial product matrices that are summed together to produce the final result.

The rising importance of memory-bound problems and SpMM in particular has induced multiple works in recent years. However, many prior works only focused on improving algorithms on multi-threaded processors [6, 5, 64] and GPUs [73, 37, 14]. Some works also explored efficient SpMM implementations on FPGAs [41, 42]. A comparative study of energy-efficient SpMM im-

plementations on contemporary platforms is done by Giefers *et al.* [22]. Prior fabricated designs, on the other hand, have only demonstrated sparse matrix-vector multiplication [16] and relatively high-density ($\geq$3%) matrix-matrix multiplication with small dimensions ($\leq$256) [2].

CHAPTER 2

Reconfigurable Hardware

2.1 Motivation

The death of Moore's law and the advent of Dark Silicon has made energy efficiency one of the most important aspects of modern architecture. While many prior designs sought to extract as much performance as possible from general-purpose cores, specialization has become a critical aspect of achieving high efficiency in cutting-edge hardware. There have been growing interest in specialized accelerators, Application-Specific Integrated Circuits (ASICs) [45], and Domain Specific Processors (DSPs) [31] over the past decades, and have been widely adopted by the industry. However, these specialized cores introduce added complexity to the programmers, who now have to navigate through various different APIs to properly utilize each special hardware. One way to mitigate this complexity is to leverage a reconfigurable core that can tweak its hardware to optimize for various different workloads, while maintain the same core API for its users [47]. Reconfigurable architecture can also quickly adapt to the rapidly changing demands of modern computing, and react faster to new, emerging workloads than designing brand new ASICs or accelerators that requires a lot of resource and time to develop. Transmuter is a coarse-grained, reconfigurable architecture designed to meet the rapidly changing demands of modern software landscape and provide programmers with the tools to tune the hardware based on the characteristics of the workload.

2.1.1 Contributions

The development of the Transmuter architecture and its implementation on gem5 simulator was done in close collaboration with Subhankar Pal and Siying Feng from University of Michigan, with Subhankar Pal being the primary architect of the hardware [53]. My main contributions in the development of the Transmuter was on the software, and the development of the prototype OuterSpace chip. The register-to-register feature in Section 2.2.4 was developed by Sung Kim [36], and Prodigy prefetcher detailed Section 2.2.6.1 is the work of Nishil Talati [69].

Subhankar and I were both primary contributors of the OuterSpace chip in Section 2.3. Subhankhar managed the overall project and developed the multiply phase with the help of Siying, while I was responsible for the development and testing of the merge phase [56]. The coalescing crossbar detailed in Section 2.3.1.2 was developed by Jielun Tan. The baseline architecture and the outer-product algorithm come from the OuterSPACE accelerator developed in collaboration with Jonathan Beaumont [52].

2.2 Transmuter Architecture

Transmuter consists of a network of General-purpose Processing Elements (GPEs) grouped together into multiple tiles, connected through a two-level cache-crossbar hierarchy to a HBM memory controller. Each tile of a transmuter contain multiple GPEs, which are managed by a Tile Manager (TM). Each GPE has their own reconfigurable L1 cache bank that are connected by a reconfigurable crossbar that feeds into a reconfigurable L2 cache bank. There is one L2 cache bank per tile, and the L2 cache banks of these tiles are connected together by a second layer of reconfigurable crossbar.

A high-level block diagram of the proposed architecture is shown in Figure 2.1. The following section describes in detail the design and functionality of each building block of Transmuter.

A real-world Transmuter-based system consists of Transmuter interfaced to an out-of-order host processor, which is responsible for execution of serial and latency-critical kernels and offloading

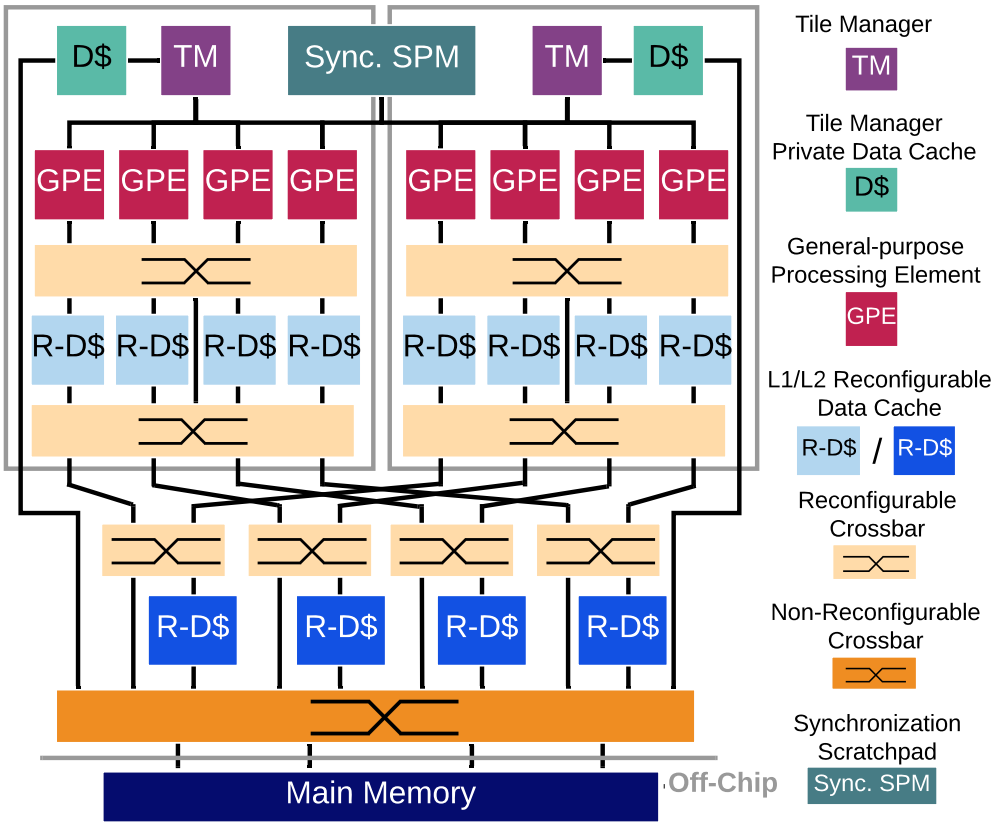


Figure 2.1: Block diagram of a 2×4 Transmuter configuration (4 GPEs/tile, 4 L1 banks/tile, 2 tiles, 2 L2 banks). The internals of the reconfigurable crossbar and cache are shown in detail, illustrating the hardware support for reconfiguration.

the rest to the Transmuter. Transmuter is thus responsible for efficient computation across a wide range of *parallelizable* kernels. A detailed discussion of the host system and host-Transmuter interface is deferred for a future work. The following sub-sections present the design and functionality of each building block of Transmuter.

2.2.1 General-purpose Processing Element and Tile Manager

The General-purpose Processing Element (GPE) is a single-issue, 4-stage, in-order Arm Cortex M-class processor, suitable for energy-efficient computation. The low-power ARM processor provides a low silicon footprint, allowing Transmuter to incorporate many GPEs in present-day reticle sizes. The GPEs support the Arm v7 Thumb ISA without SIMD instructions, and contain a single-

precision Floating-Point Unit (FPU). Minor modifications are made to the GPE pipelines to handle control hazards introduced due to a custom `PUSH/POP` interface (Section 2.2.2). The floating-point registers in the FPU were modified to provide register-to-register communication between neighboring GPEs within the same tile.

The Tile Manager (TM) is responsible for coordinating the work of all the GPEs within its tile. The TM is the same Cortex M-class of cores as the GPEs, and contains a private data cache that connects to main memory. The TMs of each tile share a synchronization scratchpad that is used to setup coherency barriers as well as any shared variables that are needed to coorderinate work across the tiles.

2.2.2 Work/Status Queues

The TMs distribute work to a the GPEs in the form of 32-bit packets that travel through a hardware FIFO *work queues* of each GPE. The GPE can respond to the messages sent through the work queue via its own *status queue* that connect to the TM through an arbiter.

The queues can be pushed or popped by using loads or stores for their designated addresses. Low-overhead decode logic translates incoming `ldr` instructions as `POP` and `str` instructions as `PUSH` commands, respectively. An important technique employed in this work to curtail dynamic power consumption in the TM and GPEs is to *block* the GPE/TM pipelines if there are structural hazards in the work/status queues. Specifically, if a work/status queue is empty and the core attempts a `POP`, the response only returns when a *producer* core pushes into the queue, and *vice versa* for a `PUSH` access. The same strategy is used for systolic array mode accesses.

2.2.3 Reconfigurable Crossbar

An $m \times n$ crossbar allows m requesters access to n resources. The implementation of a reconfigurable crossbar (R-XBar) in Transmuter is illustrated in Figure **??**. The crosspoints can be programmed to arbitrate between the requesters, support fixed connections (ON/OFF), or rotate through a series of ON/OFF patterns defined by shift registers at each crosspoint. A small control

block, the Crosspoint Control Unit (XCU), is used to program the crosspoint based on the mode, as memory-mapped I/O. The crossbars in Transmuter support the following configuration modes (Figure 2.2):

- **ARBITRATE.** When multiple requesters may attempt to access the same resource, the requests get serialized and priority is handled using a Least-Recently Granted (LRG) policy. One cycle is spent for arbitration.
- **TRANSPARENT.** Requester *i* has direct access to its corresponding resource *i*. Within a tile (L1), a GPE can access adjacent (private) memory banks with no arbitration penalty. For L2, each tile has exclusive access to one bank.
- **ROTATE.** The crossbar port connections cycle through a set of pre-programmed patterns that connect a requester and a resource. Figure 2.2 (c) shows how this configuration is used to emulate a 1D systolic array using 2 patterns (even and odd). This is extended to 4 patterns to compose a 2D systolic array. The programming model for this mode is discussed in Section **??**. The GPEs can access the corresponding memory banks with no arbitration delay.

In addition to the higher L1 R-XBar layer for GPE↔ SPM communication, a lower R-XBar layer amplifies on-chip bandwidth between the L1-RCache banks and the L2. Each L1 and L2 R-XBar has an additional "bypass" port to allow GPEs to communicate to main memory when the corresponding R-DCaches 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 entire cachelines in bursts.

2.2.4 Register-to-Register Communication

The Transmuter also allows each GPE within a tile to communicate with its neighboring GPEs through a register-to-register (r2r) communication path. The r2r queue provides a direct connection between neighboring GPEs as shown in Figure 2.3 that allow two GPEs to communicate with each other seamlessly with only a single cycle latency. When r2r queue is enabled, four floating-point

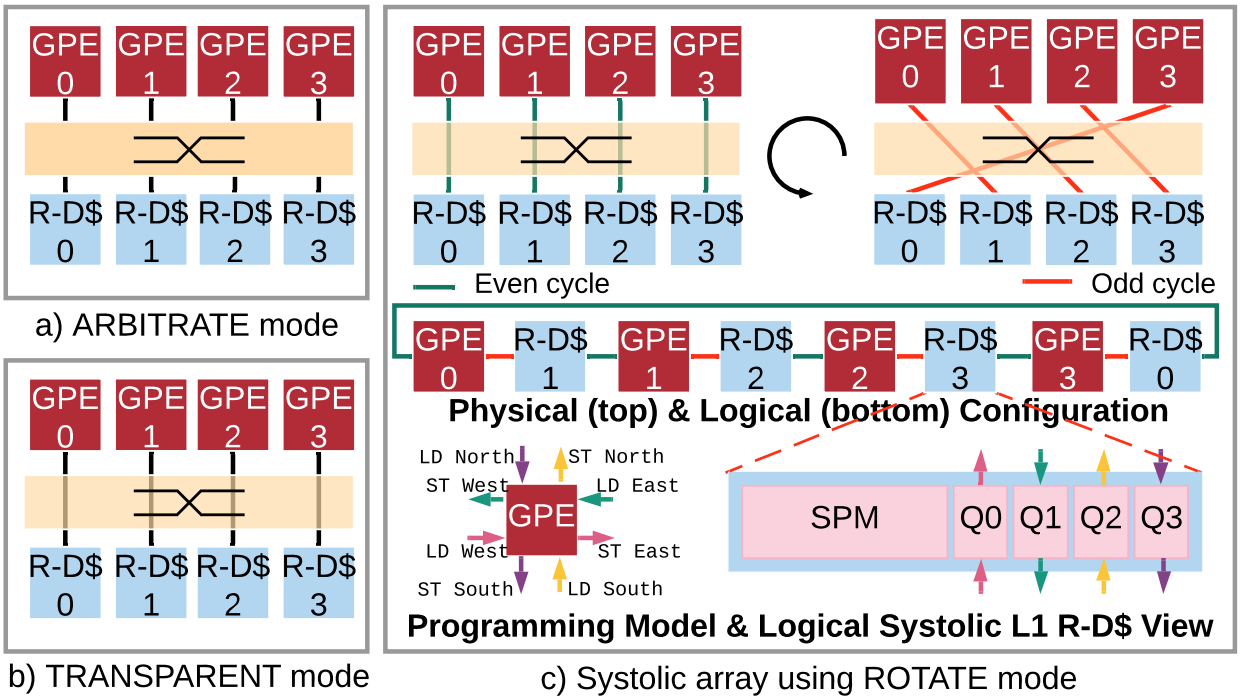


Figure 2.2: Three modes of the reconfigurable crossbar: a) ARBITRATE, b) TRANSPARENT and c) 1D systolic array configuration using ROTATE with the ports flipping each cycle. Note that b) and c) do not incur arbitration penalty. The bottom of c) shows how loads and stores are mapped to logical FIFO queue accesses `(left)` partitioned within the L1 R-DCache bank `(right)`.

registers (s0-s3 of the FPU register file) from each GPE are dedicated for the queue: one for outbound data to the left, one for inbound from the left, one for outbound data to the right, and one for inbound data from the right. Each connection is augmented with a FIFO queue to allow data buffering.

Inside each GPE, an R2R Shim that sits near the core's pipeline intercepts any writes to those FPU registers and redirect them to the the associated communication queue. Likewise, the Shim receives inbound r2r writes from neighboring cores, or gates the register file write control subject to the link state. Once the neighboring core has pushed a data onto the FIFO queue, the local reads from the corresponding r2r register proceeds normally to update the link state. Stall-based flow control logic that complements the existing control logic of the FPU pipeline prevents cores from reading state data, or overwriting unconsumed data from adjacent GPEs. The r2r queues leverage the the existing control structures to enable fast communication between the GPEs without the need of external coherence mechanisms.

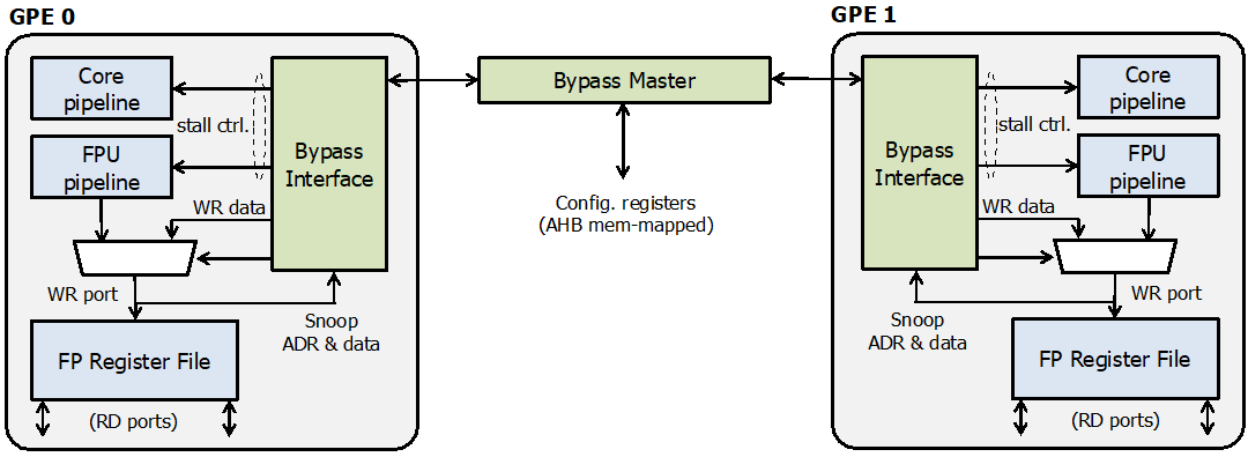


Figure 2.3: Register-to-register communication path between two GPEs.

2.2.5 Cache Configurations

The Transmuter supports three different cache configurations: cache, scratchpad, and hybrid. Cache mode is a traditional cache that can be organized to either private or shared. For L1 memory banks, shared cache mode unifies all the cache banks within each file together so the GPEs within the same tile share the first-level cache. For L2 memory banks, shared cache mode will share the cache banks of all the tiles so the whole system shares the one unified cache.

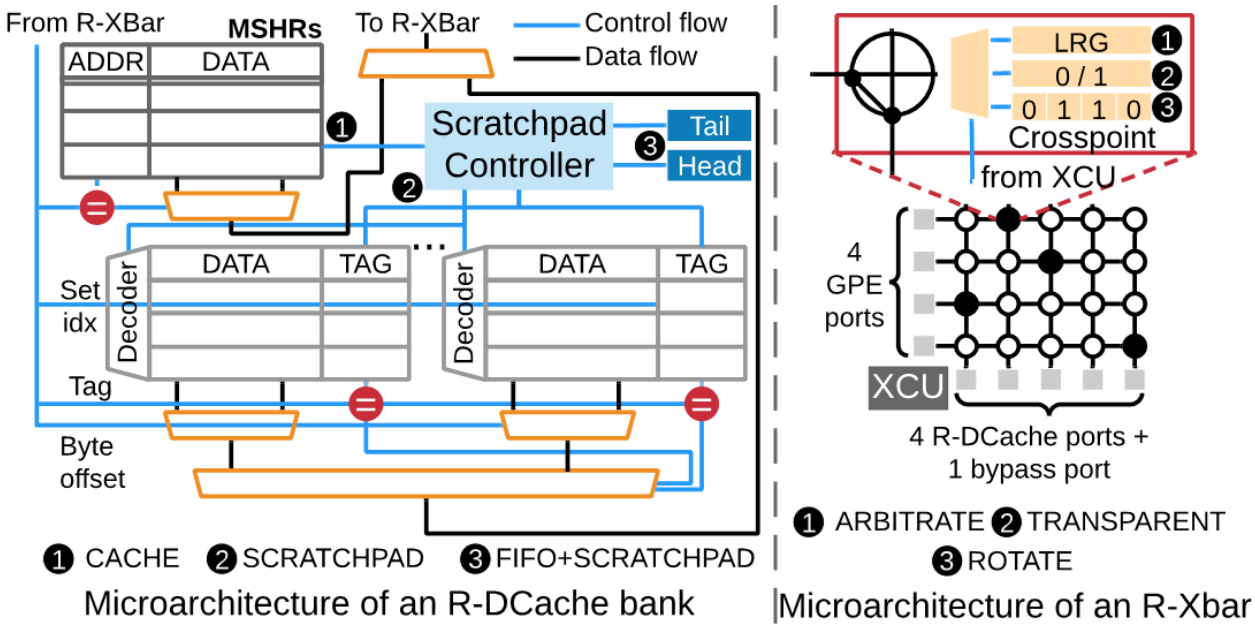


Figure 2.4: Microarchitectures of a reconfigurable data cache (left) and a reconfigurable crossbar (right) of the Transmuter.

The cache banks are configured to scratchpad mode by disabling the tag arrays and addressing the memory directly. In private scratchpad mode, the GPEs share the same address space for their own private scratchpad. In shared scratchpad mode, the individual memory banks are united together in the same way as shared cache mode, and the GPEs address the shared scratchpad through a shared address space. The scratchpad modes allow the programmer to have a dedicated memory space to store important data that needs to be accessed repeatedly during a computation, and guarantees the data to be accessed quickly. Because there is the cost of having to manage the data as well as the penalty of no longer having a cache to fallback on, scratchpad memory can provide significant speedup to a workload when used properly.

The hybrid mode is a mix of both cache and scratchpad. Unlike the two other modes that configure all the memory banks within a tile to either cache or scratchpad, the hybrid mode configures half the memory banks to shared cache, and the other half is configured to shared scratchpad. As such, the hybrid only exists as a shared mode between multiple GPEs. The scratchpad capacity in the hybrid mode is half that of a regular shared scratchpad. The hybrid mode seeks to provide a middle ground for workloads that benefits from scratchpad memory, while still providing access to a cache to minimize the performance degradation for parts of a workload that do not utilize the cache.

2.2.6 Memory Prefetcher

Prefetching is an important part of mitigating memory latency. A memory prefetcher uses the pattern of recent memory accesses to predict future memory fetches and loads them onto the cache beforehand. Transmuter utilizes a simple stride prefetcher, which upon seeing a memory access from a GPE fetches a block of nearby data into the cache. The stride prefetcher takes advantage of the spatial locality of a workload to increase the cache hit rate.

Typical prefetcher focuses on predicting the future accesses of a workload and preemptively filling the cache, but does not synergize with any of the scratchpad modes of the Transmuter. A programmable prefetcher that fetched data directly onto the scratchpad and offer a variety of

different prefetching mode was prototyped on the OuterSpace prototype chip as detailed in Section

.

2.2.6.1 Prodigy Prefetcher

Prodigy is a low-cost hardware/software co-design solution for intelligent prefetching [69]. The prefetcher is designed to improve the memory latency irregular workloads, including sparse linear algebra, graph analytic, and fluid mechanics that exhibit two specific types of data-dependent memory access patterns. It incorporates static program information from software, in addition to dynamic run-time information from hardware. The key component of the prodigy prefetcher is the Data indirection graph (DIG)—a proposed compact representation used to express program semantics such as the layout and memory access patterns of key data structures.

First, a compiler extracts program semantics and encodes them encoded as a DIG. The encoded DIG that is inserted into the application binary, is utilized by a low-cost hardware prefetcher to fetch data according to the algorithm's traversal pattern. By dynamically adapting the prefetch distance to an application's execution pace, the prefetcher is able to remain flexible and stay up-to-date with the execution of the problem.

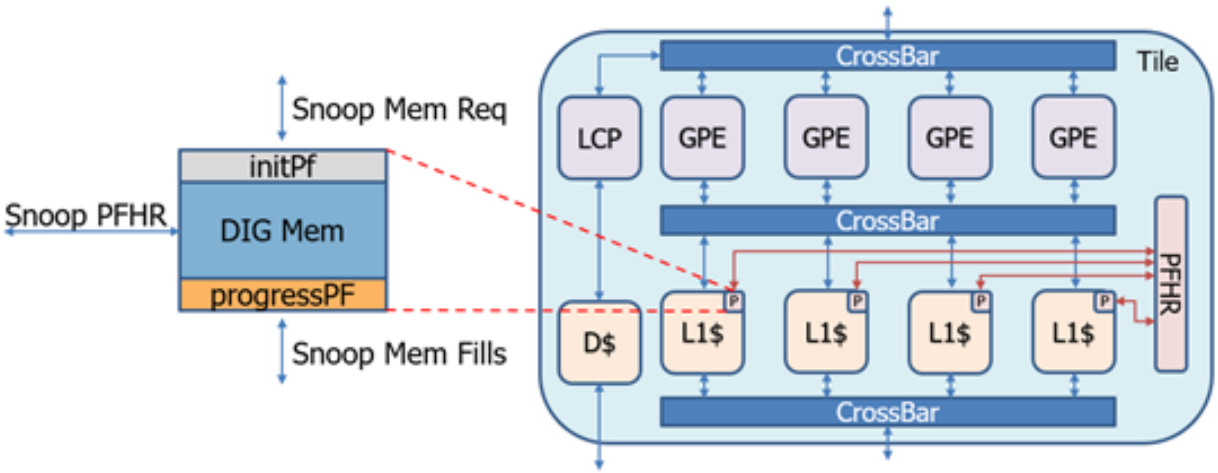


Figure 2.5: Overview of Prodigy Graph Prefetcher on Transmuter.

Figure 2.5 shows a diagram of how the prodigy prefetcher is implemented on the Transmuter. Each cache bank of a GPE is armed with its own prefetch engine. There are two key features: (i)

The prefetch engine monitors the memory fills from the GPEs as well as the response from the lower memory hierarchy, (ii) To support shared cache mode, the prefetch status holding register (PFHR)s, which contain the status of the prefetcher progress, are fused together so that they can be accessed by every prefetch engine.

In a typical prefetcher, only the higher-level memory requests get monitored for calculating the prefetch addresses. However, prodigy prefetcher monitors memory fills from the lower-level memory and issue further prefetch requests based on the data stored in specific addresses. This is done by having the prefetcher generate following requsets if the target data is already in the cache. If the data is not present, the prefetcher is notified once the current prefetch requests are filled with the corresponding data. In addition, since the Transmuter supports both shared and private cache modes, the PFHRs are fused together within a tile and connected to each prefetch engine within the tile. The prefetch engine then accesses the corresponding PFHRs based on the current cache mode.

2.3 Outer-Space Accelerator

Outer-Space is a sparse matrix-matrix multiplication (SpMM) accelerator that served as a prototype chip for the Transmuter. The chip was designed specifically to accelerate SpMM workloads using the use of outer product-based approach.

Parts of the text and graphs for this section were taken from prior publications on IEEE Journal of Solid-State Circuits [54, 56].

2.3.1 Architecture

The top-level diagram of the outer-space architecture is shown in in Figure 2.6. Outer-space consists of two compute substrates. The first, composed of 32 PEs (4 PEs/tile), computes the *multiply* phase. Each PE has a 32-bit Floating-Point (FP) multiplier and supports out-of-order loads/stores. The second substrate consists of eight ARM Cortex M0+M4 pairs (1 pair/tile) for the *merge* phase.

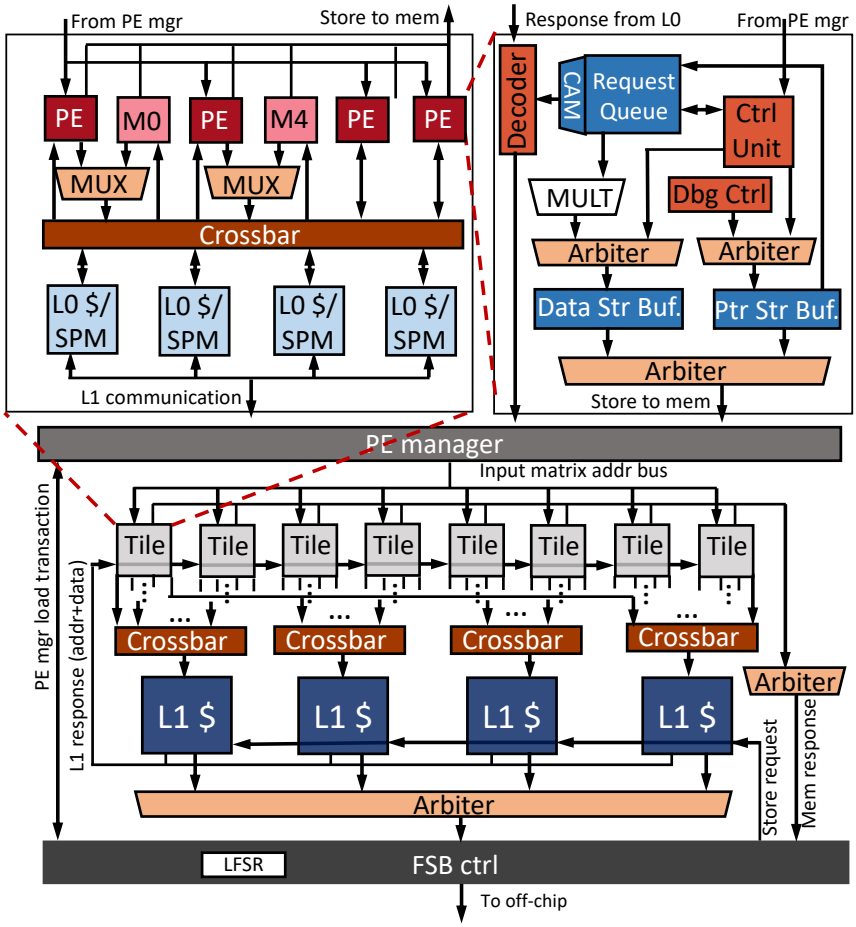


Figure 2.6: Top level diagram of the outer-space microarchitecture, a tile, and a PE. The chip contains a total of eight tiles, with each tile consisting of four PEs and a pair of M0 and M4 cores.

All the compute elements are connected through a reconfigurable network. The network consists of a fully-synthesizable Swizzle-Switch Network (SSN) crossbar based on [63], with the original pull-down networks replaced by OR trees (Figure 2.7). The synthesizable SSN still uses the same priority algorithm, but can also be easily ported to different process technologies since it does not require a custom layout. The crossbars support request coalescing, multicasting (Figure 2.9) and Least-Recently Granted (LRG) arbitration (Figure 2.8).

2.3.1.1 Compute Units

The processing elements (PEs) are custom Finite State Machine-based elements that perform the *multiply phase* of the outer product algorithm. At the core of the PE is a Control Unit (CU) that

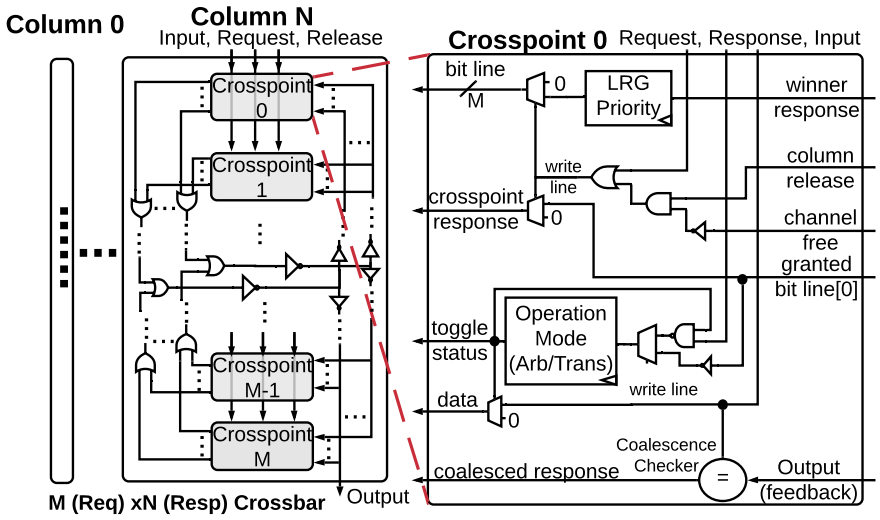


Figure 2.7: OR Trees of the Swizzle-Switch Network (SSN) crossbar. Each Crosspoint is one requester and the bitwise OR'd results are sent back to each Crosspoint.

walks through the algorithm state machine. The CU initiates loads of elements of columns of Matrix A and rows of Matrix B, tracking requests in a *request queue*. The *request queue* is a structure that allows *out-of-order* loads to the elements of the input matrices. Load responses satisfy an entry in the queue by associatively searching the address field of each *request queue* entry. Each PE also houses a single-cycle, single-precision floating point multiplier that multiplies elements of A and B as soon as they are available in the *request queue*. The calculated partial product elements are stored into a "data" store buffer. This is a simple FIFO queue of (address, data, valid) tuples. There exists a separate buffer to store pointers, which is associatively-searchable, unlike the data buffer. Through this split store buffer design, we are able to reduce the energy consumed by limiting expensive associative searches to fewer registers. Finally, a debug block is used to relay important messages at programmable intervals to the off-chip interface, such as state of each PE, number of multiplications committed, etc.

The general purpose cores, Arm Cortex-M0 and Cortex-M4 cores, handle the computation in the *merge* phase. They are both low-power, in-order cores designed for high energy efficiency. The M4 performs the bulk of the computation including the floating-point operations. The M0 acts as a programmable prefetcher for loading data into the scratchpad independent of the M4's operation.

The M0 and M4 cores communicate through the use of local scratchpad memory for shared data, and hardware mutex locks to streamline synchronization. The mutex locks come in two types: first-come-first-serve (FCFS) mutex and sleep mutex. The FCFS mutex is a simple synchronization lock where the core that acquires the lock first prevents the other core from acquiring the lock, until the first one releases it. When querying the lock for acquisition, the cores have the option to stall until the lock is freed. The sleep mutex is a unidirectional lock with a predetermined owner. Sleep mutex begins with its lock pre-acquired by its designated core, and the non-designated core stalls whenever it accesses a locked mutex. During the *merge* phase, the sleep mutex is used by the M4 core to prevent M0 core from starting the prefetch before M4 has finished initiating the metadata.

2.3.1.2 Coalescing Crossbar

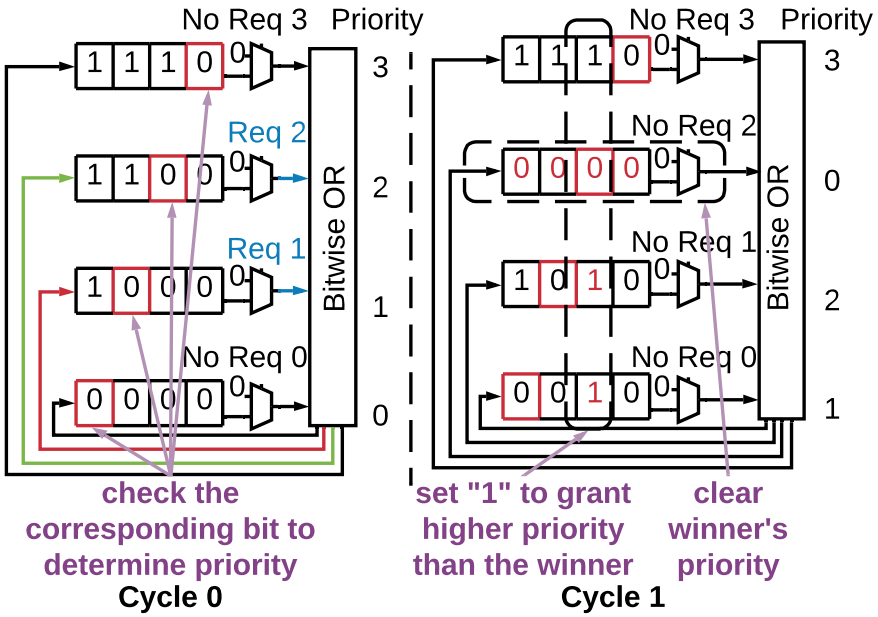


Figure 2.8: Least Recently Granted (LRG) Scheme. The requesting nodes assert their priority bits and the winner is determined based on which requester receives a 0 in the corresponding response bit. The winner's priority bits are then cleared.

The crossbar takes one cycle to arbitrate, based on a least-recently granted (LRG) scheme, and another cycle to transmit data. As shown in Figure 2.8, each requester sends its priority bits to be bitwise OR'd. The corresponding bit of the result vector, based on the index of the requesters, is

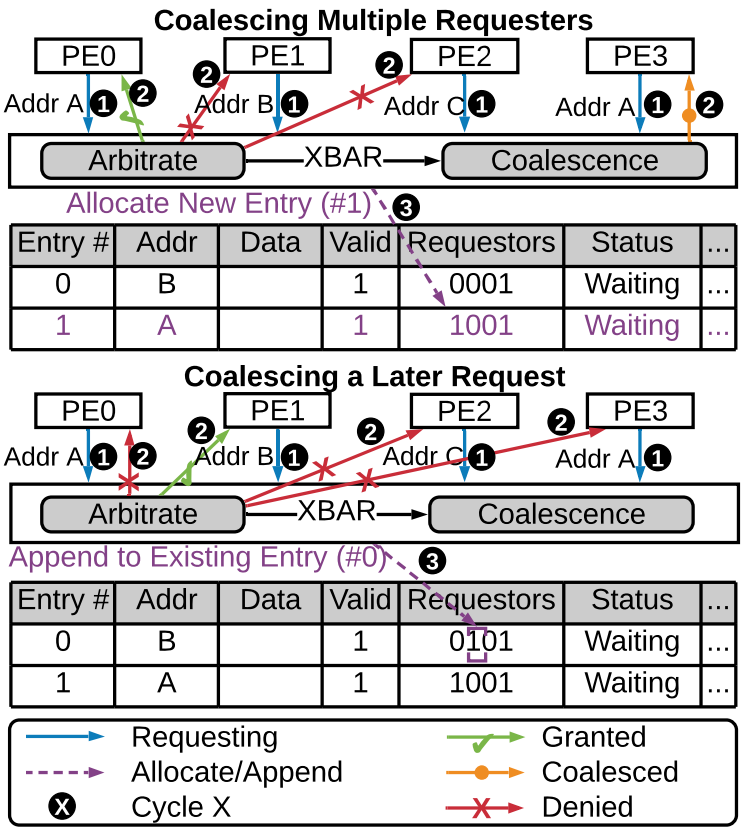


Figure 2.9: Crossbar and cache coalescence. The crossbar coalesces identical requests by marking the requesters in a bit vector, which is then stored in the cache controller. While it is in the cache controller, more requests can be coalesced along the way should there by any requesters asking for the same address.

sent back to the requesters and the one with a 0 on its granted bitline wins. Next cycle, the winner clears its priority bits and other requesters set the priority bit corresponding to the winner to 1, granting them higher priority than the winner. In any particular cycle, one column will always be zero among all requesters, since there will always be one with the highest priority. If any channel is not actively requesting, it will assert all 0s instead of its actual priority bits to put it on the lowest priority possible. For example in Figure 2.9, in Cycle 0, only requesters 1 and 2 request the channel, and therefore only these two assert their priority bits while 0 and 3 assert all zeroes. The result of the bitwise OR would be 1100, and then each requester checks their corresponding bit, in which case requester 2 wins. Since requesters 0 and 3 did not request, it ignores the result of the bitwise OR. The winner, in this case requester 2, then clears its priority bits. Once granted,

the requester can hold on to the channel until it chooses to free the channel. Requests can be coalesced in the crossbar, shown in Figure 2.9. Since the channel can observe all the requesters and their requesting addresses, it can simply compare them with the winner's address and grant to any matching requesters. Coalescence does not affect the priority status, since it happens after arbitration.

2.3.1.3 Reconfigurable Cache

The downstream L0 crossbar connects to the reconfigurable L0 cache consisting of four logical SRAM banks, each of which consists of four physical SRAM banks. The L0 cache provides second-level coalescing by comparing the new requests with existing pending requests stored in the miss status holding registers (MSHR). Along with tracking missed requests, the MSHRs also act as a request queue that takes in the inbound requests, a fill buffer that temporarily holds the returned data before storing to SRAM and a response queue that sends the read data back to the PEs. For coalescence, each MSHR entry stores a bit vector of all requesters and adds additional requesters, should any coalesce in the process. The upstream crossbar then multi-casts the read data back to the PEs based on the requester bit vector.

For the *multiply* phase, the L0 is a multi-banked set-associative cache, allowing NZEs of B to be shared. For *merge*, it is reconfigured into a multi-banked scratchpad by disabling the tag array and the Least-Recently-Used (LRU) counter, and is private to each M0-M4 pair. Through another set of coalescing crossbars, the L0 cache in each tile connects to the L1 layer, which interfaces to the front side bus (FSB).

Only minor modifications were made to the cache controller to enable reconfiguration into scratchpad mode. In the scratchpad mode, the tag arrays and the set index bits are disabled, and the controller addresses directly into each SRAM bank.

2.3.2 Algorithm Mapping

In the *multiply* phase, each Processing Element (PE) multiplies an element of a column of the first operand (A) with a row of the second operand (B). The row elements of B are reused across all the PEs and thus are stored in an on-chip shared cache (Figure 2.10-a). For the *merge* phase, we switch to a pair of Arm cores, Cortex-M4 and Cortex-M0, connected by private on-chip memory. The two cores act as a single unit to stream in the results of *multiply*, perform *mergesort*, and store the final results to DRAM. Our studies reveal that a scratchpad leads to better performance than a cache for this phase, due to the irregular nature of data accesses.

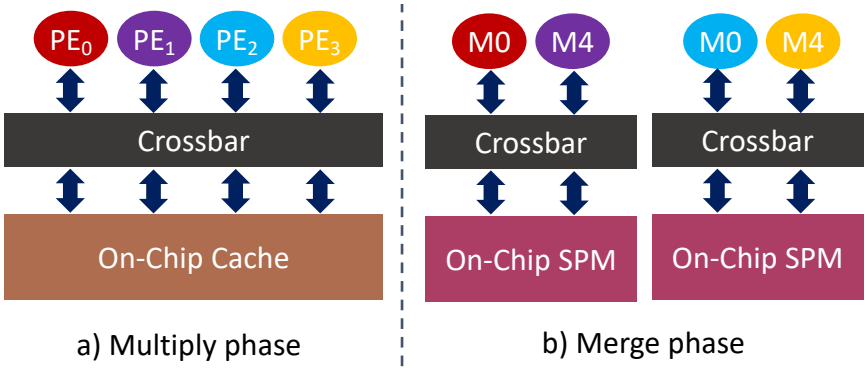


Figure 2.10: High level overview of architecture suited for the different phases of outer product-based matrix-matrix multiplication. Shared cache is suited for the predictable patterns in the *multiply* phase, whereas a private scratchpad is better for *merge*.

2.3.2.1 Outer Product Algorithm

In this section, we describe in detail how the outer-product algorithm is mapped on to the hardware. The outer product approach consists of two phases: the *multiply* phase and the *merge* phase. The *multiply* phase sweeps through the input operands, A and B, to perform all the multiplication operations and generate the intermediate Partial Product Matrices (PPMs). The *merge* phase consolidates these PPMs into a single result matrix C. The *multiply* phase is performed by custom PEs using a shared cache configuration for on-chip memory, while the merge phase is processed by M4 ARM cores with the memory reconfigured to be a private scratchpad for each core.

In the *multiply* phase, each Processing Element (PE) multiplies a non-zero element of column i of A with all non-zero elements of row i of B to produce one Partial Product Matrix (PPM) row. Each NZE is fetched only once. The PPMs are stored as a set of linked lists of pointers to "chunks" in the DRAM, as shown in Figure 1.2. The *multiply* phase computes multiplications of *all* combinations of fetched elements, resulting in maximum reuse of inputs *without* any index matching, thus circumventing the problem of unproductive loads. Since each PE traverses through the non-zero elements of a row in Matrix B, the memory access during this phase is sequential and predictable. In addition, multiple PEs operate on the same row for each column element that corresponds to the row, resulting in high data reuse across the PEs.

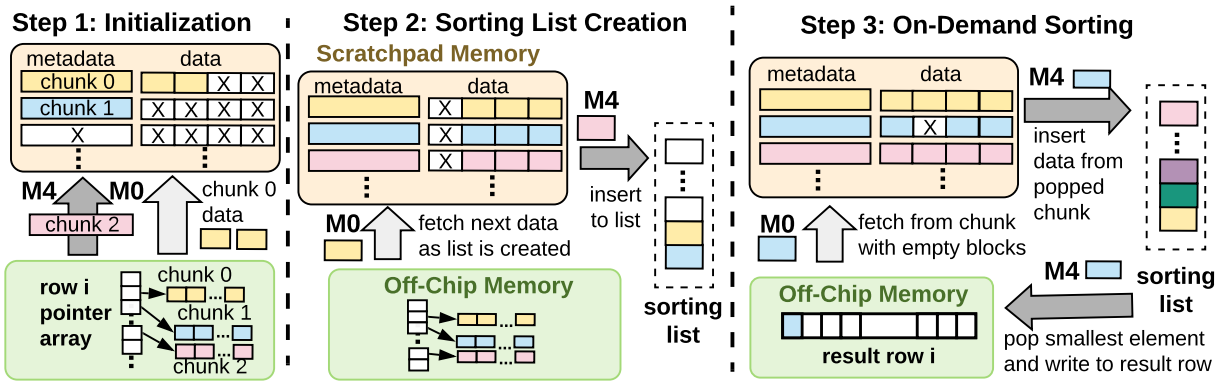


Figure 2.11: Breakdown of the three steps of *merge* phase on Outer-Space accelerator: initialization, sorting list construction, and on-demand sorting. M4 core performs the sorting operation on the data that has been loaded into the scratchpad by the M0 core.

In the *merge* phase, each M4 core is assigned a pointer array of chunks that correspond to a single row of the result matrix C. When merging the different chunks, the M4 core needs to ensure that all the elements in the final row are *ordered by their column index*. To ensure this ordering, each M4 core maintains a sorting list. The sorting list only needs to be big enough to hold one element from every chunk that is being merged by this core. This is because all the chunks are ordered by their column indices when they are produced in the *multiply* phase. Once the first element of every chunk is inserted into the sorting list, the steady state involves writing out the smallest element to DRAM and fetching one element to be sorted. The element with the smallest

column index in this sorting list is also the element with the smallest column index within the pool of PPM rows that need to be merged. By inserting the next element of the PPM row that the last smallest element came from, there can be a constant pipeline of smallest elements while keeping the sorting list size the same.

The *merge* phase, as shown in Figure 3.6, is broken down into three steps: **initialization**, **sorting list construction**, and **on-demand sorting**.

Step 1. Initialization Each chunk is augmented with metadata that is used to keep track of the number of elements that have been fetched by the core. During initialization, the metadata of each chunk assigned to the core is written into the scratchpad memory.

Step 2. Sorting list construction The M4 core begins constructing the sorting list by inserting the head of every chunk into the list (Step 2a). The list is sorted again each time an element is pushed into the list, based on the column index. The core iterates over all of its assigned chunks, and so the list starts with first element of every chunk. As the M4 core inserts elements from the scratchpad memory into the sorting list, the M0 core fetches the next elements of the chunk into the new empty blocks (Step 2b).

Step 3. On-demand sorting The M4 core *pops* the smallest element of the list to be placed in the output buffer (Step 3a). The M4 core checks the chunk that this popped element originated from, fetches the next element in the chunk, and *pushes* it into the sorting list (Step 3b). The popped element is compared against the element that is currently in the output buffer. If the indices of the two elements match, the values of the two elements are summed. If the indices do not match, the element in the output buffer is written to memory as the first element of one row in the result Matrix C. The popped element then becomes the new element in the output buffer, and the next element is fetched from the chunk of the last popped element. This process is repeated until all the assigned chunks have been processed. As the M4 core consumes data from the scratchpad memory, the M0 core independently fetches the next data of each chunk onto the emptied blocks (Step 3c).

Unlike the *multiply* phase where most of the memory accesses are sequential and there is plenty of data sharing between different PEs, the data accesses of the merge phase are mostly irregular,

with no shared data across the Arm core pairs. Each core is assigned a disjoint pool of chunks, so that each Arm core pair operates on independent memory space. The location of each memory load is determined by the element that was popped from the sorting list. Therefore, the memory access is highly irregular and difficult to predict. Because the two phases have such drastically different access patterns, we implemented a reconfigurable architecture that can tune its memory hierarchy based on the needs of each phase.

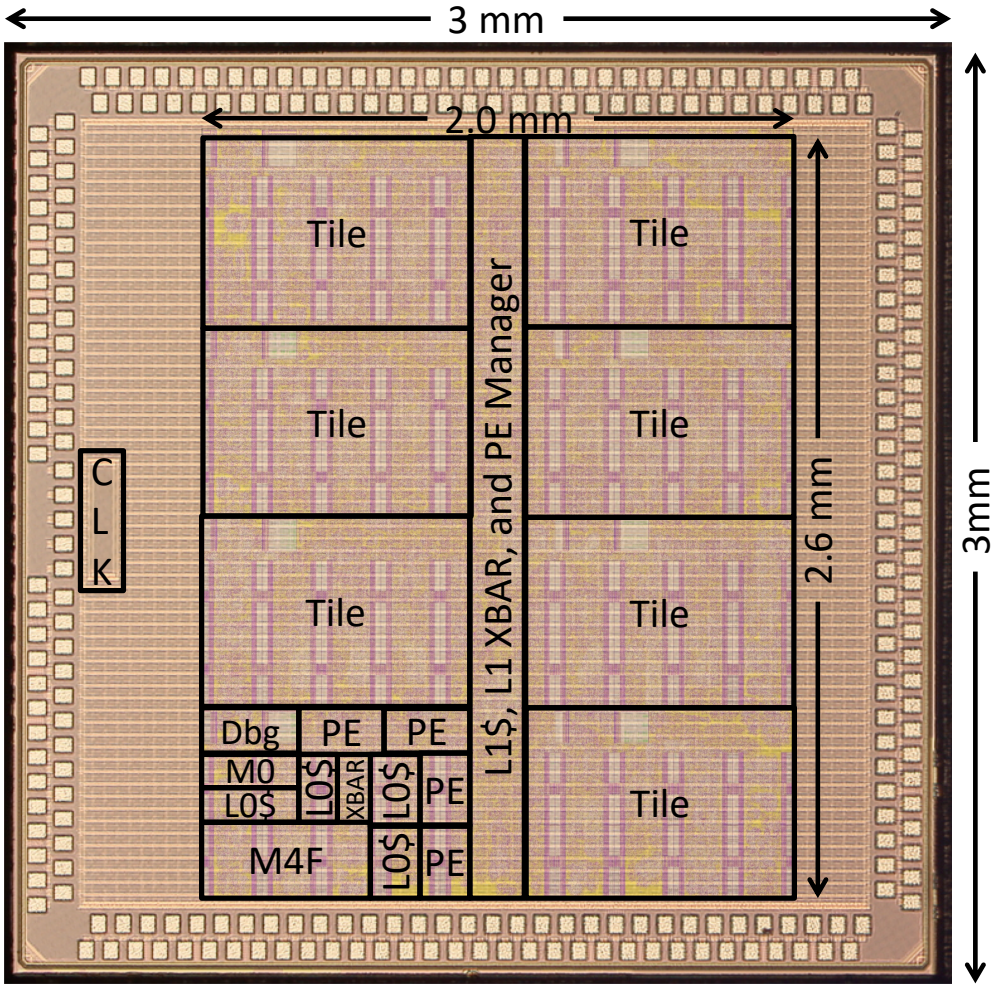


Figure 2.12: Annotated 3.0 mm $\times$ 3.0 mm die photo of outer-space accelerator with GDS overlay. There are eight tiles per chip, each tile containing an ARM Cortex-M0, a Cortex-M4, and four PEs.

2.3.2.2 Scratchpad Prefetching

While all the core computation of the *merge* phase is handled by the M4 cores, each M4 is paired with an M0 core (Figure 2.10), which acts as a programmable prefetcher. The primary purpose of the M0 core is to fill the private scratchpad with the elements of the PPM rows, so that the M4 core can grab its data from the scratchpad instead of the memory.

The M0 starts fetching the head elements of the chunks at the initialization step. M0 begins fetching data once the metadata of a chunk has been registered into the scratchpad memory. This allows the M4 core to immediately proceed to the construction of its sorting list, without waiting on the memory. As the M4 core pushes a new element from the scratchpad into the sorting list, the M0 core loads the next element of the chunk into the evicted space, until all the elements have been consumed.

Due to the size of the local scratchpad, there is a limit to the number of chunks that can be held in the scratchpad. This also limits the length of the sorting list maintained by the M4, since the length of the sorting list is equal to the number of individual chunks being merged. When the total number of chunks assigned to an Arm core pair exceeds the maximum length of the sorting list (L), the PPM rows are divided into subgroups of L PPM rows. The merge phase is then performed in multiple passes, each one generating an intermediate result of L merged chunks. During each pass, the intermediate results are written out to a temporary space in memory. Once there is enough capacity to merge the remaining chunks as well as the intermediate results, the final merge pass produces a single, fully merged row of result matrix C. These intermediate passes are expensive because the data needs to be stored in external memory, and read again during the final merge pass. To minimize the number of passes, L needs to be as high as possible. However, for the M0's prefetching to be effective, each chunk needs to have sufficient number of elements that have been loaded ahead in the scratchpad. Therefore, there exists a trade-off between the number of PPM rows that is tracked during the merge phase, and the number of elements that can be prefetched into the scratchpad for each PPM row.

2.3.3 Power and Performance

The performance of 2.0 mm×2.6 mm outer-space accelerator for Sparse Matrix-Matrix multiplication, with the chip layout shown in Figure 2.12, was evaluated through matrix squaring on synthetic matrices, as well as power-law graphs that are representative of real-world sparse matrices [8], [62]. The measured characteristics of the chip are summarized in Table 2.1.

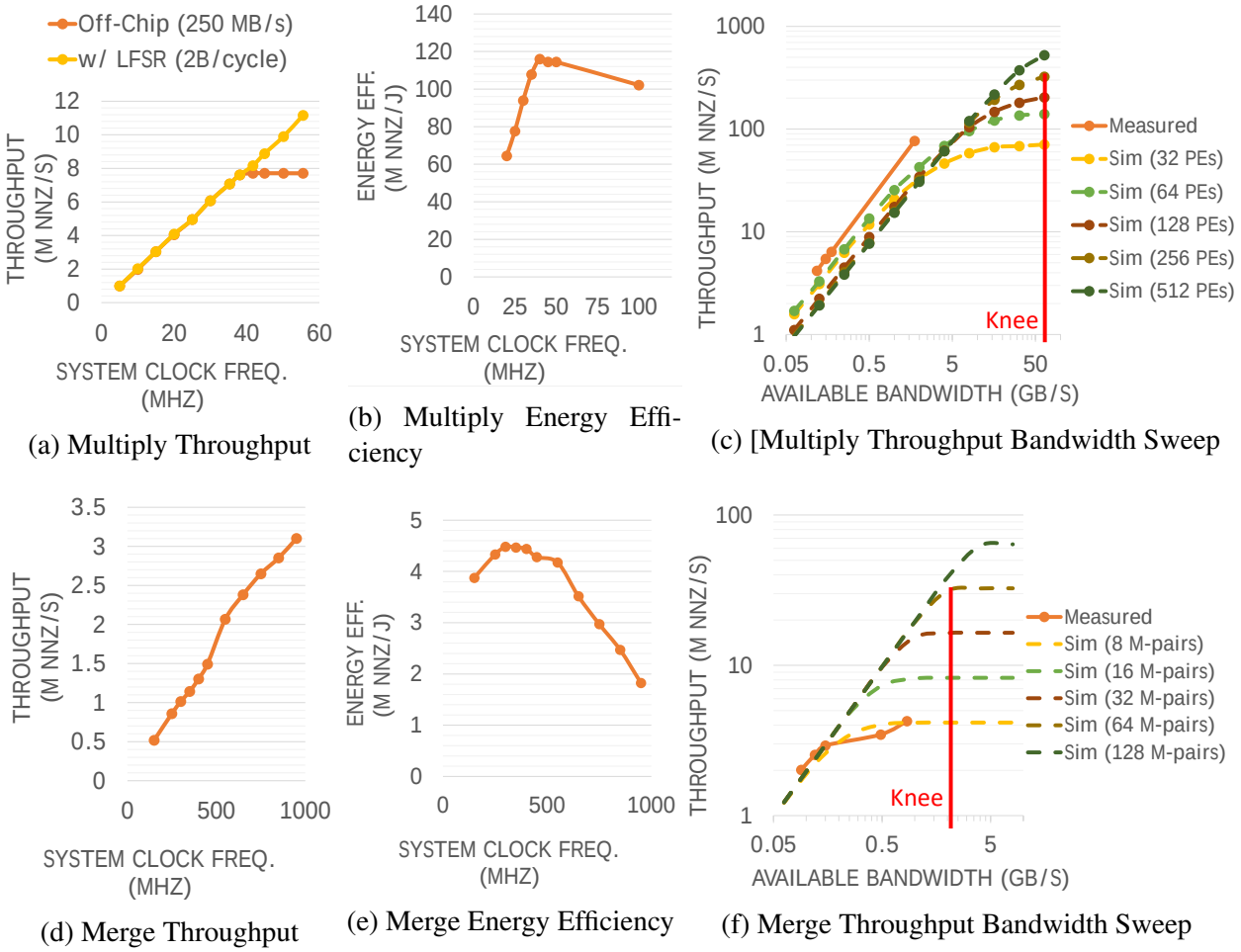
At the optimal frequency and voltage points, the accelerator achieves an energy efficiency of 6.1-8.4 M NNZ/J and bandwidth efficiency of 6.4-15.5 M NNZ/GB. The SSN crossbar gives the chip a 24.9% performance gain at 86.3% the energy and 1.3% more area over a MUX crossbar based design.

Table 2.1: Chip characterization summary

Technology	40 nm CMOS
Die Size	3.0 mm × 3.0 mm
Block Size	2.0 mm × 2.6 mm
# Transistors	25,134,927
Total SRAM	112 KB
Data Precision	Single-Precision Floating Point
Nominal Frequency (Minimum Energy)	41.7 MHz 0.860 V (Multiply) 352.0 MHz 0.864 V (Merge)
Maximum Frequency	950.0 MHz 1.27 V
Nominal Power Consumption	66.6 mW (Multiply) 226.0 mW (Merge)

2.3.4 Frequency and Bandwidth Sweep

Figure 2.13 shows the clock and bandwidth sweeps for a matrix of size 100K×100K and density of 0.0008%. The *multiply* and the *merge* phases were evaluated separately in order to determine the optimal parameters for each phase. Clock sweeps show that while *multiply* performance hits a roofline, *merge* performance saturates slowly, as *merge* is more compute-heavy due to the overhead of maintaining the sorting list. We observe the frequency and voltage level in which the chip achieves optimal energy efficiency to be at 41.7 MHz and 0.860 V for the *multiply* phase, and 352.0 MHz and 0.864 V for the *merge* phase.



(a) Multiply Throughput

(b) Multiply Energy Efficiency

(c) [Multiply Throughput Bandwidth Sweep

(d) Merge Throughput

(e) Merge Energy Efficiency

(f) Merge Throughput Bandwidth Sweep

Figure 2.13: Clock and bandwidth sweeps for matrix dim. 100k, density 0.0008%. For measurements with increased bandwidth, an on-chip LFSR is used for multiply and the M0 is used for merge.

For the bandwidth sweeps, simulation results are appended to measured results to illustrate the impact of higher bandwidth and more compute units. The performance of *multiply* phase continues to increase with higher bandwidth, while the *merge* phase reaches saturation early, at less than 1 GB/s. The "knee" lines show that the *multiply* phase is $\sim$30$\times$ more sensitive to bandwidth than the *merge* phase.

Based on the frequency and bandwidth scaling of the chip, scaling out our current chip to 16$\times$ the current configuration would meet the CPU's performance at 9.5$\times$ less bandwidth, 16.7$\times$ lower power and 0.08$\times$ the area. At this configuration, the chip will be able to make optimal use of available bandwidth by minimizing off-chip traffic.

2.3.5 Benefits of Reconfigurable Memory

One of the key design choices of the chip is the use of reconfigurable memory that transitions between *cache* and *scratchpad* based on the demands of the algorithm. For workloads with well-defined data access and reuse patterns, the scratchpad improves performance over the cache by preventing any data that will be reused by the program from getting evicted out to memory during intermediate computation, ensuring each critical data to only be fetched once. Figure 2.14 shows the benefit of using the scratchpad memory during the *merge* phase at varying matrix densities, but with the matrix dimension fixed. We observe average performance benefit of 25.7% across the different matrices, with higher benefits for denser matrices.

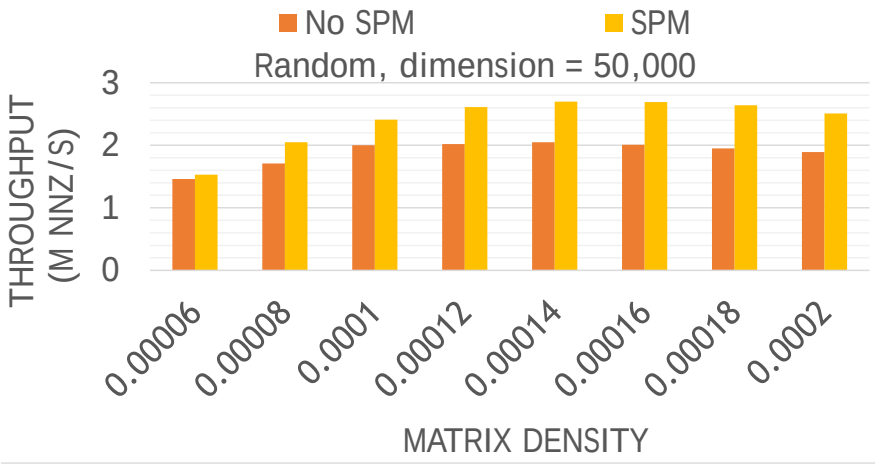


Figure 2.14: Merge phase performance with and without scratchpad memory. Overall performance benefit of scratchpad is 25.7%.

CHAPTER 3

Sparse Linear Algebra

3.1 Motivation

Linear algebra algorithms are the fundamental building blocks of many workloads that are used in today's computing [25]. From various scientific computing to image processing and artificial intelligence, linear algebra operations can be found in a wide variety of different fields. With the rise of machine learning in recent years, linear algebra kernels have gained even greater importance for modern computing platforms. Traditionally, linear algebra computing was focused on accelerating dense matrix and dense vector workloads, and this have been achieved through the development of wide, highly parallel vector machines. However, emergence of cloud data and large graph data has introduced new challenges in the form of big, sparse datasets.

Traditional, linear algebra algorithms focused primarily on dense data; matrices consist of mostly nonzero values. Processing these dense matrices requires great amount of compute power to process the tasks, as well as memory bandwidth that can sustain the high data throughput. However, as compute power and storage capacities continued to improve exponentially in modern times, these matrices have become larger and larger. Modern graph datasets originating from social networks or cloud services can easily surpass over a million nodes [65]. While data grew larger, their density decreased as most of these large matrices were sparse matrices; matrices that contain very few nonzero values. This meant many of the existing algorithms that were optimized for dense data were inefficient, as any computation involving the zero values of a sparse matrix were

wasted computation [17, 13, 23, 64]. Therefore, in order to efficiently compute these big, sparse datasets, one must apply sparse algorithms specifically designed to process sparse data, and the underlying hardware must also be designed to complement the unique challenges of accelerating sparse algorithms [28, 19].

Matrix-Matrix multiplication and matrix-vector multiplication are two fundamental building blocks of linear algebra solutions, and take up a large portion of modern computing efforts. Existing architectures cannot adequately meet the demands of these two kernels in the context of sparse data, as traditional CPUs and GPUs were designed to optimize dense data algorithms. Thus, we focused our efforts on improving these two key kernels for very large, highly sparse matrices.

3.1.1 Contributions

The initial idea of outer-product based sparse matrix-matrix multiplication was developed in close collaboration with Subhankar Pal and Jonathan Beaumont [52]. I lead the development and analysis of the merge phase and the different variations, as well as the row-wise approach. The inner product algorithms for sparse matrix-matrix multiplication and sparse matrix-vector multiplication used for benchmark comparison were developed by Siying Feng [53, 19]. Yuhan Chen provided the pre-processed matrices for the matrix re-ordering.

3.2 Matrix-Matrix Multiplication

Matrix-matrix multiplication is a key kernel in graph algorithms such as breadth-first search, as well as other linear algebra algorithms. In contrast to linear algebra operations involving dense matrices, sparse matrix computations are harder to accelerate on traditional hardware such as a CPU or a GPU. Sparse matrices, by definition, contain a large amount of irrelevant data (zeros) that result in a large amount of wasted resources when computed using traditional dense matrix algorithms such as inner-product matrix-matrix multiplication. There are three main ways to compute matrix-matrix multiplication: row-column (inner-product), column-row (outer-product) [7, 72],

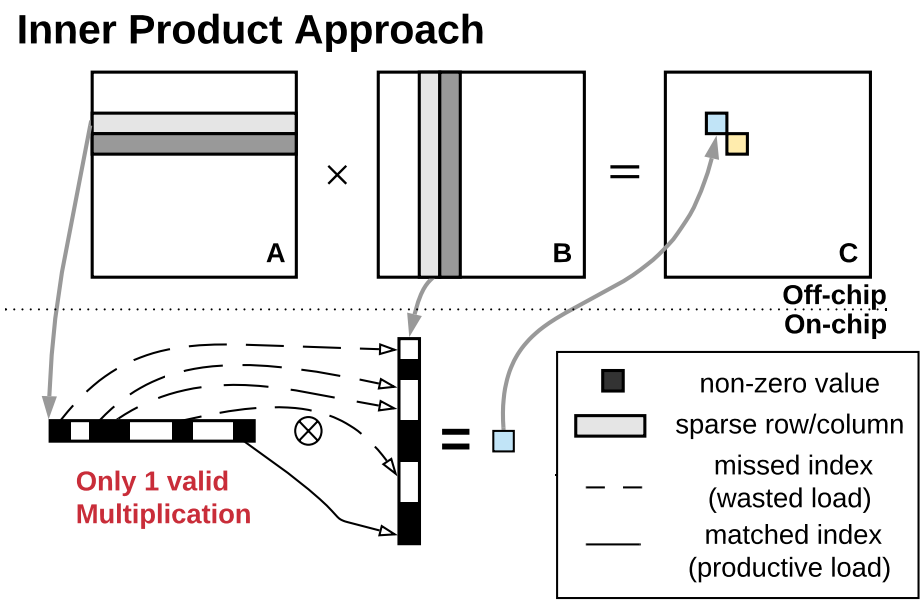


Figure 3.1: Overview of inner-product approach to matrix-matrix multiplication. Rows of matrix A are multiplied by columns of matrix C to produce the elements of matrix C.

and row-row (rowwise) [66].

3.2.1 Inner-Product Multiplication

In inner-product multiplication, each row of matrix A performs a vector multiplication with each column of matrix B to produce each element of the result matrix C, as shown in Figure 3.1. The inner-product multiplication is the traditional method commonly employed in dense matrix multiplication. However, this method can be highly inefficient when computing with sparse matrices, as the nonzero elements of each row and columns needs to have matching index in order to be multiplied together to generate the partial products. When the input matrices are sparse, the inner-product method wastes significant amount of computation time performing index matching during the multiplication process, making this algorithm perform poorly.

Inner-product matrix-matrix multiplication is mapped onto the Transmuter by assigning a row of matrix A to each GPE. Each GPE takes its assigned row vector and performs a vector multiplication with each of the column of matrix B. The resulting vector multiplication is written directly

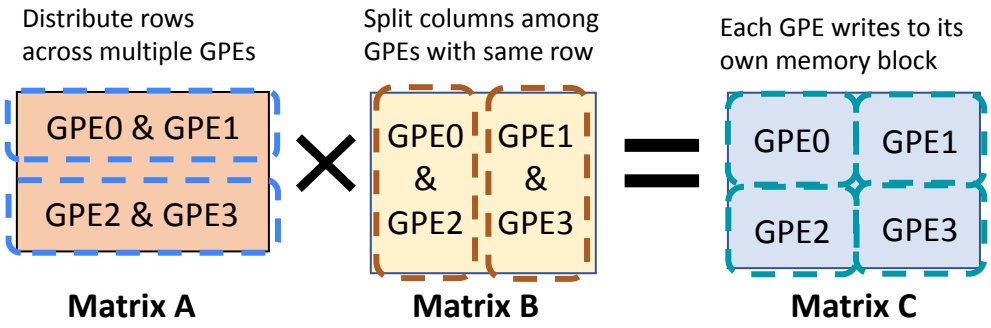


Figure 3.2: Diagram of inner product matrix-matrix multiplication mapping onto the Transmuter GPEs with data blocking.

to memory, and the GPE proceeds to the next available vector pair. To take advantage of the spatial locality and maximize data reuse, blocking is used to limit the number of columns of matrix B each GPE operates on, as shown in Figure 3.2. Rather than having all the GPEs fetch every single column of matrix B, each GPE only needs to focus on a handful of vectors from matrix B that can fit into their local cache. This strategy is especially useful when operating on private scratchpad mode or the hybrid mode, where there is heavy restriction on the amount of available scratchpad memory.

3.2.2 Outer-Product Multiplication

In the outer-product method, each column of matrix A are multiplied with corresponding row of matrix B to generate a partial product matrix of the result matrix C, as shown in Figure 3.3. Once all the partial-product matrices are computed, they are all merged together to produce the result matrix. While the outer-product method performs poorly for dense matrix inputs due to the high memory overhead of the partial-product matrices, when the matrix is sparse, the algorithm can be more efficient than the inner-product method because the multiplication can be performed between the nonzero elements without the need of any index matching. In addition, with the outer-product method, the nonzero elements of each input matrix only need to be accessed once.

The outer-product method is split into two execution phases: the multiply phase and the merge phase. In the multiply phase, each nonzero element in a column of matrix A are multiplied to the

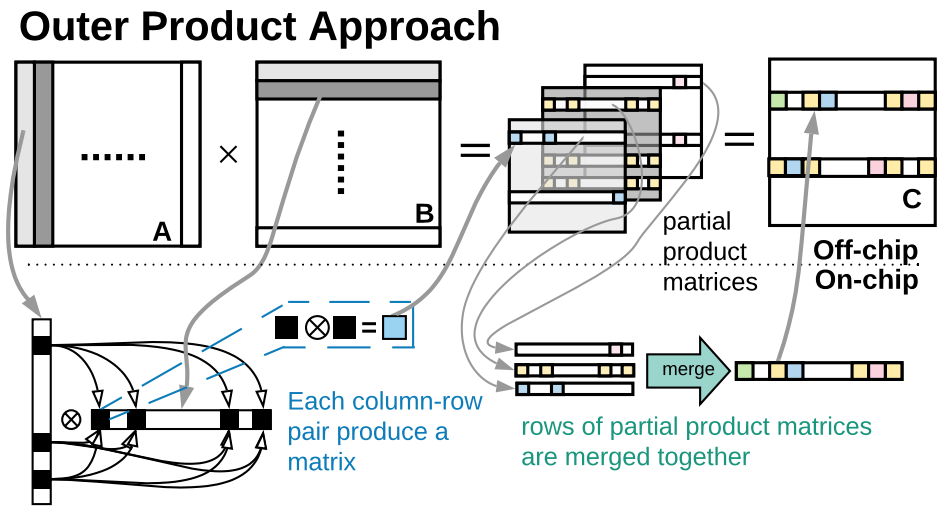


Figure 3.3: Overview of outer-product approach to matrix-matrix multiplication. Columns of matrix A are multiplied by rows of matrix C to produce partial-product matrices, which are merged together to generate the result matrix C.

corresponding row of matrix B to produce the partial-product matrix. These partial-product matrices need to be stored in memory, as they cannot be utilized until the multiply stage is complete. The core workload of multiply phase is fairly straightforward, so the main point of concern in this phase is the work distribution.

Once all the columns of matrix A are processed, the execution moves onto the merge stage. In the merge phase, the partial-product matrices that were generated need to be merged together to produce the final matrix C. Because the partial-product matrices are stored in sparse format, the nonzero elements need to be kept in order of the column indices throughout the merging process. Three different merging algorithms were explored for the outer-product method: dense vector merge, sorting list merge, and systolic merge.

3.2.2.1 Static vs Dynamic Multiply

Two different work distribution strategies were explored for the multiply phase: static and dynamic. In static multiply, work is distributed evenly across all the processing elements. The number of column elements assigned to each GPE is fixed, keeping the work management simple and

lightweight. Dynamic multiply allocates work to the GPEs as they become available. For each column element of matrix A, the LCP checks for a GPE with a free work queue and assigns the work to that GPE. This mitigates any load balancing issues that arise from some column-row pairs taking more time than others during the static allocation, leading to load imbalance issues where other GPEs stay idle for a significant period of time while waiting for the slowest GPE to complete its execution.

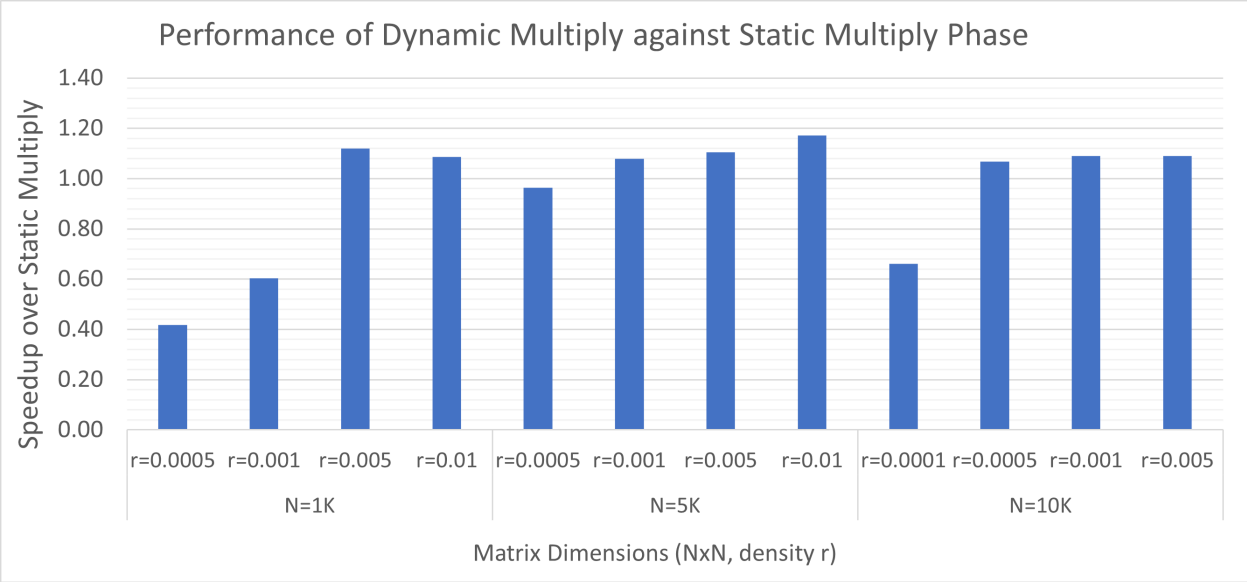


Figure 3.4: Speedup of the execution time of the dynamic work allocation against the static allocation in multiply phase for outer-product multiplication. Dynamic allocation exhibits performance improvement of up to 17% for matrices with density greater than 0.5%, while static allocation shows better performance when density is too low.

Figure 3.4 shows how the performance of dynamic work allocation compare against the static allocation. The sparse matrix-matrix multiplication tested was $A * A^T$, where matrix A is a square matrix of sizes N =1K, 5K, and 10K, and density ranging from $r = 0.0005$ to $r = 0.01$. Dynamic work allocation outperforms the static allocation by up to 17% for matrices of density greater than $r = 0.5\%$, thanks to better load balancing. On the other than, static allocation performs significantly better than dynamic allocation for very sparse matrices. This is because these matrices have too few nonzero elements in each of its rows and columns, that the extra overhead of the dynamic allocation hinders execution more than it benefits.

3.2.2.2 Dense Vector Merge

Dense vector merge is the most naive implementation of the merge phase. In this implementation, a dense vector is used to accumulate the rows of partial-products to produce a single row of the final matrix C. as shown in Figure 3.5. Each row of partial-product that correspond to the same row of the final matrix are fetched and accumulated onto a dense vector. Because the partial-products are accumulated into a dense vector, the values can be directly indexed into the corresponding positions in the dense vector. Once all the partial-products have been processed, the resulting dense vector is converted into sparse format to be a row of matrix C. This process is iterated until all the rows of matrix C are computed. Because the dense vector needs to be the same size as the row of matrix C, this implementation scales poorly with large matrices. However, when the size of matrix C is relatively small, *dense vector merge* can outperform all other merge implementation thanks to its low overhead.

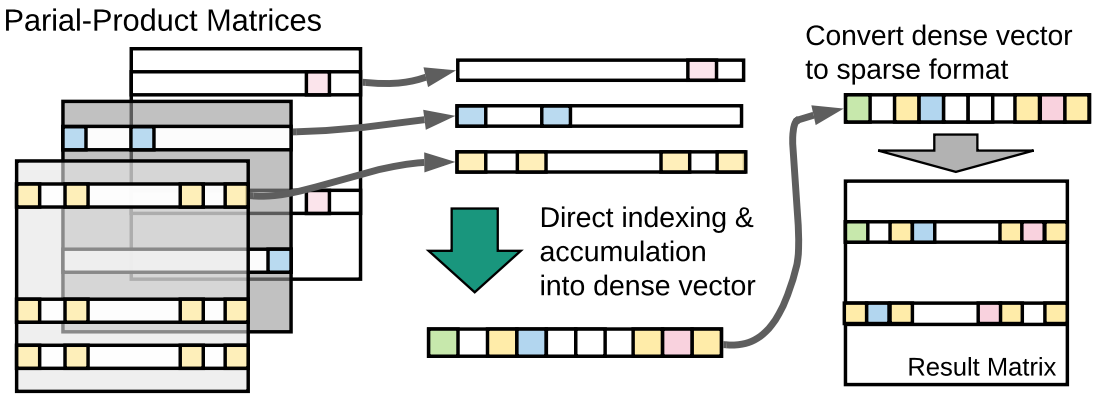


Figure 3.5: Diagram of the dense vector merge. The rows from the partial-product matrices index directly into the dense vector and accumulate with the data. Dense vector needs to be converted to sparse format before writing to memory.

3.2.2.3 Sorting List Merge

In the *sorting list merge*, the partial-products are merged using a list sorted based on each element's column index that is constantly maintained throughout the execution. The main challenge of merging the partial-product rows is keeping track of their column indices so they are merged into the

final row in the correct order. In addition, any elements with matching indices need to be summed together. To achieve this, a sorting list consisting of one element from each partial-product row is maintained.

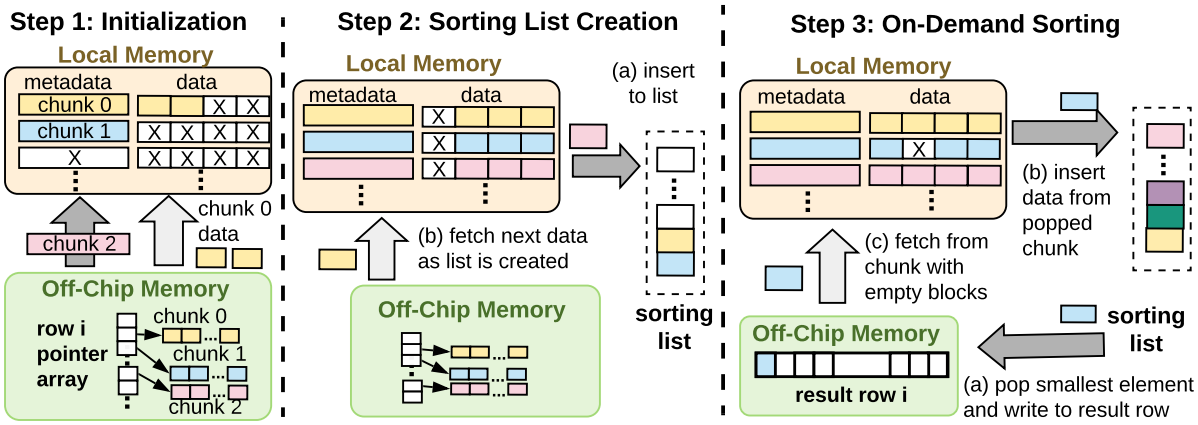


Figure 3.6: Breakdown of the three steps of *sorting list merge* for outer-product Matrix-Matrix multiplication: initialization, sorting list construction, and on-demand sorting.

Sorting list merge is broken down into three steps: **initialization**, **sorting list construction**, and **on-demand sorting**.

Step 1. Initialization Each partial-product row (chunk) is augmented with metadata that is used to keep track of the number of elements that have been fetched by the core. During initialization, the metadata of each chunk assigned to the core is written into the local memory.

Step 2. Sorting list construction Processing elements begins constructing the sorting list by inserting the head of every chunk into the list (Step 2a). The list is sorted again each time an element is pushed into the list, based on the column index. The core iterates over all of its assigned chunks, and so the list starts with first element of every chunk. If N is the number of chunks associated with a row of the result Matrix C, then the maximum size of the sorting list of each GPE is N.

Step 3. On-demand sorting The processing element *pops* the smallest element of the list to be placed in the output buffer (Step 3a). The processing element checks the chunk that this popped element originated from, fetches the next element in the chunk, and *pushes* it into the sorting list

(Step 3b). The popped element is compared against the element that is currently in the output buffer. If the indices of the two elements match, the values of the two elements are summed. If the indices do not match, the element in the output buffer is written to memory as the first element of one row in the result Matrix C. The popped element then becomes the new element in the output buffer, and the next element is fetched from the chunk of the last popped element. This process is repeated until all the assigned chunks have been processed.

Because each partial-product row is ordered in increasing column index, as long as one element of each partial-product row is entered into the sorting list, it is guaranteed that the smallest element of that sorting list is the smallest element of all the partial-product rows. Once the smallest element is found, that element is popped and written into the corresponding row of matrix C, and the proceeding element of the partial-product row that the popped element came from is pushed into the sorting list. Whenever the index of the popped element matches the index of the previously popped element, the two elements are accumulated together. Iterating this process until all the elements of the partial-product rows have gone through the sorting list will produce the correct final row.

3.2.2.4 Sorting algorithm selection

One of the biggest challenges of sorting list merge is the scalability of the underlying sorting algorithm. The complexity of the sorting and maintain the sorting list increases with the number of partial-product rows. Because the merge phase is only interested in the smallest element within list, two different designs were explored for the sorting algorithm: linear and priority-queue.

Simple Linear Sorting utilizes an one-dimensional array of size N as the sorting list structure. Whenever an element is inserted into the list, the list is traversed one-by-one from the beginning until the correct position for the new element is found, as shown in Figure 3.7. When the new element is placed in position, all the subsequent elements of the list has to be shifted by one position, and the size of the list is increased by one. The complexity of this inserting operation is $O(N)$.

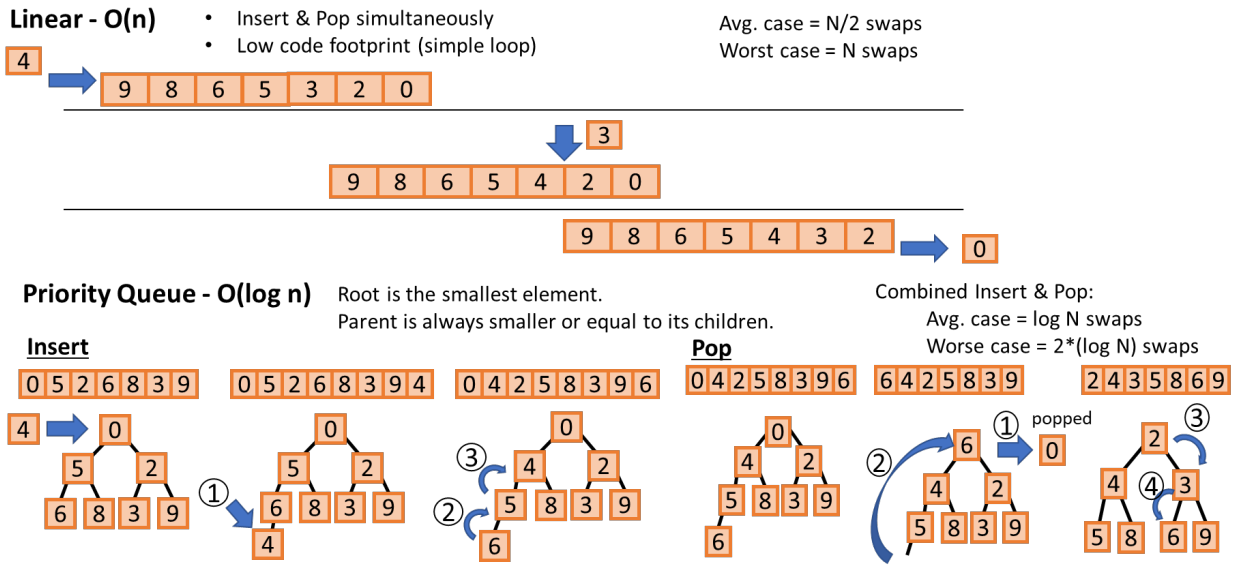


Figure 3.7: Overview of the two sorting algorithms: linear and priority queue. Linear algorithm is simple, but has a complexity of $O(N)$. Priority queue requires more computation, but can perform pop and insert in $O(log(N))$.

The sorting list is maintained in decreasing order, so ”popping” the smallest element from the list can be performed without any reorganization of the data structure by simply decreasing the size of the list. Thus, the popping operation can be completed in $O(1)$ complexity. The linear sorting algorithm is the simplest to maintain, and has low overhead for small lists, but the cost of accessing and maintaining the list increases linearly with the size of the list.

Priority-Queue Sorting maintains the sorting list as a priority queue structure as exhibited in Figure 3.7. The priority queue is a balanced binary tree structure where the root is always the smallest. In this min-heap structure, the parent node is always smaller or equal to both its children. The functions for popping the smallest element and insertion are shown in Algorithm 1.

The smallest element of a priority queue is always the root, so the complexity of finding the minimum is always $O(1)$. However, the popping and re-balancing operation requires traversing down the tree once in the worst case, so the complexity of popping op eration is $O(log(N))$. Similarity, the insertion operation also requires the tree to be traversed from bottom up in the worst case, so the complexity is $O(log(N))$. Thus, the combined complexity of popping the smallest element then inserting a new element that is required during the merge phase is $O(log(N))$. Compared to

Algorithm 1 Priority Queue Sorting List Functions

```
function PQUEUEPOPSMALLEST(target)
    Remove root node
    Move the last node to root
    PQueueFixDown(new root)
end function
function PQUEUEFIXDOWN(target)
    if target > smallestChild(target) then
        Swap(target, smallestChild)
        PQueueFixDown(target)          ▷ Keep moving down the tree until it's in the right position
    end if
end function
function PQUEUEINSERT(target)
    Append target to end. It is now the new last node.
    PQueueFixUp(last)
end function
function PQUEUEFIXUP(target)
    if target < parent(target) then
        Swap(target, parent)
        PQueueFixUp(target)            ▷ Keep moving up the parents until it's in the right position
    end if
end function
```

the $O(N)$ complexity of the simple linear list, the priority queue scales better with larger data.

Although the computational bottleneck of the sorting list can be mitigated using the priority-queue structure, there is still the problem of memory overhead of maintaining such large lists. In order for the sorting list merge to perform efficiently, the sorting list needs to be stored in fast, reliable memory such as a scratchpad. However, for very large matrices, it is possible for the size of the sorting list to exceed the capacity of the scratchpad. Two solutions were explored to mitigate the scratchpad capacity problem: spill-over memory and the multi-pass merge.

Spill-over memory allocates space in the off-chip memory to be used once the scratchpad capacity is exceeded. Once the spill-over is triggered, the parts of the sorting list that do not fit in the scratchpad are now stored in memory, until the size of the sorting list is reduced to be contained entirely in the scratchpad. While the execution flow remains unchanged, there is a considerable degredation in performance because the processing unit needs to switch back and forth between scratchpad memory and off-chip memory.

Multi-pass merge breaks down the partial-product matrices into multiple batches that can fit

in the capacity of the scratchpad. These individual batches are merged together in separate passes to consolidate each batch into a single partial-product matrix each. Once the number of remaining partial-product matrices is reduced to a number that can fit in the local scratchpad, the final pass merges all the partial-products together to a single final matrix. While the multi-pass merge allows the sorting list merge to be conducted even with limited scratchpad memory, the cost of triggering a multi-pass process is quite significant, as each pass generates a new partial-product row that needs to be written to memory and then read back into the scratchpad for the final merge.

3.2.2.5 Systolic Merge

One of the main bottlenecks of the sorting list merge is scalability. While priority queue structure and multi-pass merge help mitigate the problems that arise from the sorting list growing too large, the penalty of operating on lists that have outgrown the capacity of the local scratchpad is quite considerable. These issues all stem from the merge operation of each row being constrained to a single processing unit. Systolic merge tackles this issue by distributing the merge workload across multiple processing units.

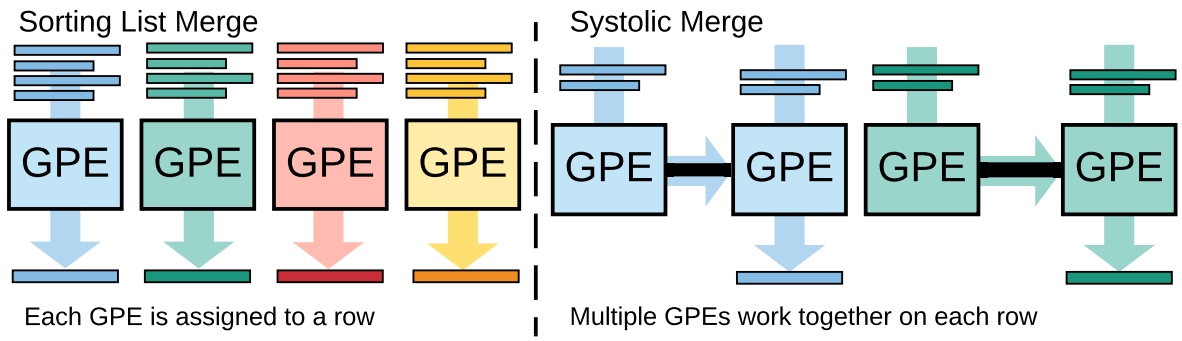


Figure 3.8: Diagram of how work is allocated to the GPEs in sorting list merge vs systolic merge. Systolic merge splits the merge workload amongst the GPEs and use the register-to-register connection to communicate between the GPEs.

In the systolic merge, the merging of a single row is distributed across multiple processing units that are connected in a systolic manner, as shown in Figure 3.8. The partial-product rows are distributed evenly across these processing units, and each one maintains its own sorting list in a

similar manner as the sorting list merge. However, instead of writing the smallest element of the sorting list to memory, the processing units pass their popped elements to the next processing unit in the systolic chain. Each processing element receives the popped element and pushes it to its own sorting list, treating the elements that gets passed down in the same manner as a partial-product row that it needs to merge together. This way, the distributed partial-product rows are being processed in parallel in smaller batches, while the sorting order is still maintained as a whole through the systolic link. The last processing unit in the systolic chain writes its results as the final sorted row of matrix C. A register-to-register connection is used to achieve efficient transfer of data between the neighboring processing elements. The underlying algorithm of systolic merge is similar to the sorting list merge, but the main difference comes from how the work is distributed across the GPEs. Thus, when the number of GPEs assigned per row is 1, then the systolic merge is identical to the sorting list merge.

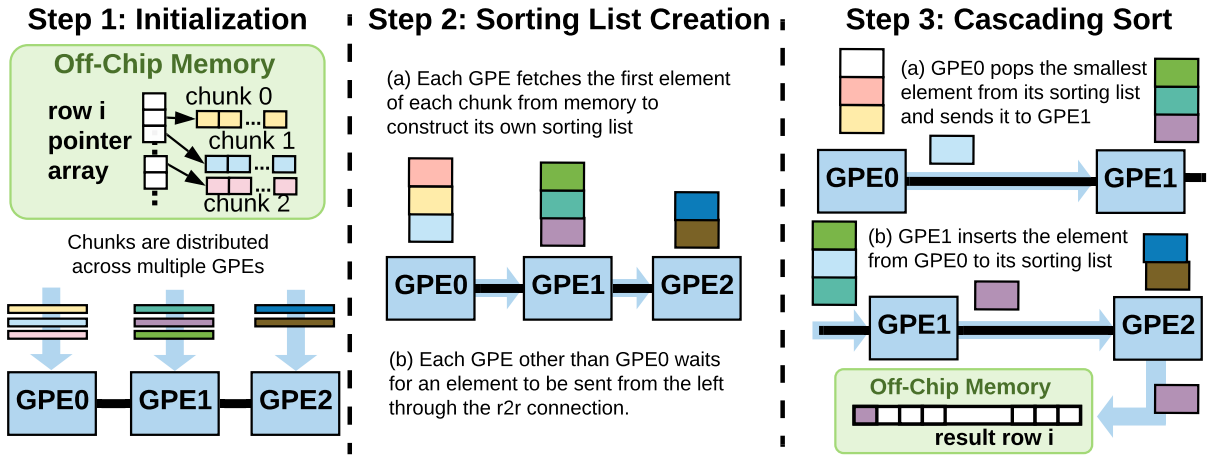


Figure 3.9: Breakdown of the three steps of *systolic merge* for outer-product Matrix-Matrix multiplication: initialization, sorting list construction, and cascading sort.

Similar to sorting list merge, the systolic merge is broken down into three steps as shown in Figure 3.9: **initialization**, **sorting list construction**, and **cascading sort**.

Step 1. Initialization The partial-product rows (chunks) corresponding to the same row of final matrix are distributed amongst the processing elements that share the same systolic link. Work is

distributed more heavily towards the earlier GPEs because the execution is limited by the latter GPEs.

Step 2. Sorting list construction Each processing element begins constructing the sorting list by inserting the head of every chunk assigned to itself into its sorting list (Step 2a). The list is sorted again each time an element is pushed into the list, based on the column index. The processing elements iterate over all of its assigned chunks, and so the list starts with first element of every chunk. When one element from every chunk has been inserted into the list, each processing element also proceeds to fetch an element from the processing element on its left through the register-to-register connection (Step 2b). Each GPE treats the data coming from the left as an additional chunk.

If N is the total number of chunks corresponding to a given row of matrix C and the number of GPEs assigned to each row is K, then on average each GPE is assigned $\frac{N}{K}$ chunks, and the sorting list is $\frac{N}{K} + 1$.

Step 3. Cascading sort The left-most processing element pops the top of its list and pushes popped element to its right (Step 3a). The subsequent processing elements take the element that has been passed down from the left and insert it into its own list. The processing element *pops* the smallest element of the list and pushes it to the right (Step 3b). The processing element checks the chunk that this popped element originated from, fetches the next element from a chunk or a neighboring processing element, and *pushes* it into the sorting list. When the right-most processing element pops an element from its list, the popped value is written to the row of the result matrix C. This process is repeated until all the elements in each chunk has been processed.

Unlike the dense vector merge and the sorting list merge that assigns only one processing element to each row being merged, systolic merge distributes the merge workload across multiple processing units, reducing the compute and memory loads of an individual GPE. This helps the algorithm better scale with both larger data, as well as denser ones.

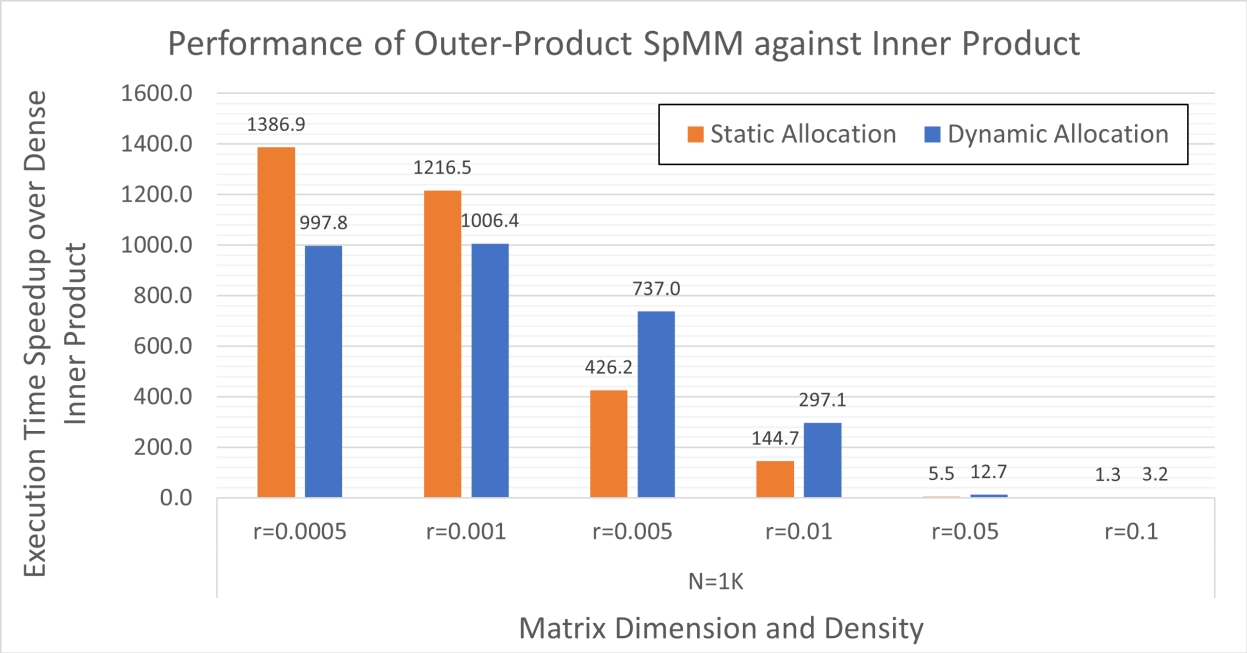


Figure 3.10: Performance of outer-product based sparse matrix-matrix multiplication against dense inner product algorithm for square matrices of size $N = 1$K, and density from $r = 0.05\%$ to 1%. Outer-product exhibits speedup of up to 1386x, with static allocation outperforming dynamic allocation the sparser the matrix.

3.2.2.6 Performance Analysis

Performance of the sparse outer-product algorithm against a vanilla dense inner-product is shown in Figure 3.10. The two different multiply allocation schemes were tested for matrix of size $N = 1$K, and density from $r = 0.05\%$ to 1%, with the merge set to a simple sorting-list merge. The inner-product algorithm is a simple dense algorithm that performs the same for all densities. The outer-product sparse matrix-matrix multiplication outperforms the inner-product by up to 1386x, with the performance gap decreasing as the matrix gets denser. While static allocation has better performance than dynamic allocation when the matrix is very sparse, dynamic allocation starts to outperform static allocation density $r \geq 0.005$.

Figure 3.11 shows the performance comparison of outer-product method using dynamic work allocation against static allocation over a wider range of matrix size and densities. The two allocation schemes with sorting list merge were compared across uniform random square matrices of $N = 1$K to 10K, with density ranging from $r = 0.0001$ to 0.01. While dynamic allocation outper-

forms static allocation by up to 2.24x, when the matrix is too sparse, static allocation leads to better performance. This is due to the high overhead of dynamic allocation being too costly for highly sparse matrices that have too few work to distribute across multiple GPEs. Since static allocation exhibits up to 28% improvement in performance for some matrices with density less than 0.01%, a flexible scheme that switches between static and dynamic allocation based on the known density of the input matrix could be the most optimal.

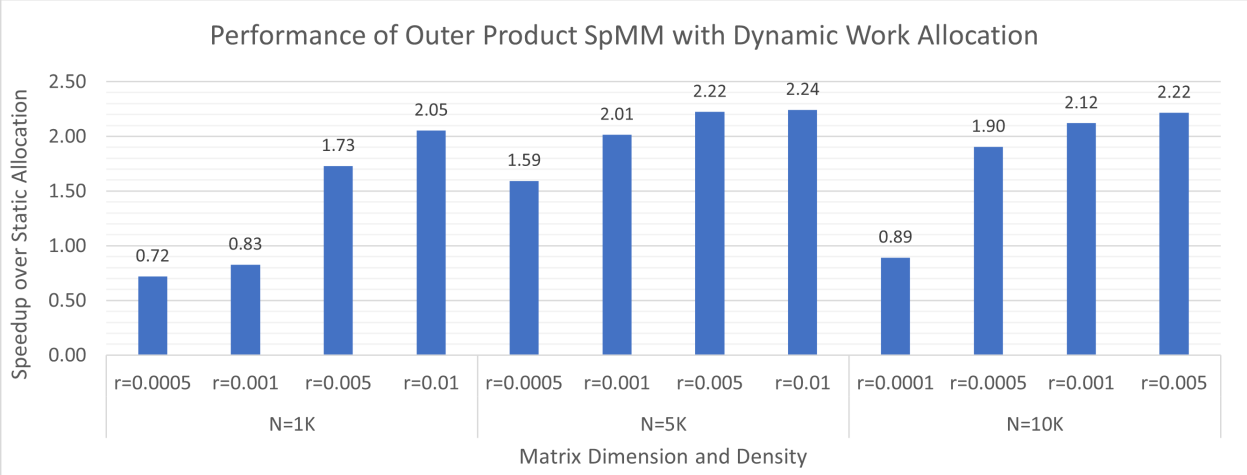


Figure 3.11: Performance of outer-product based sparse matrix-matrix multiplication with dynamic work allocation against static work allocation for uniform random square matrices of size $N = 1$K to 10K, and density from $r = 0.01\%$ to 1%. Dynamic allocation outperforms the static allocation by up to 2.24x.

The performance of systolic merge was compared against the sorting list merge in Figure 3.12. The execution time of systolic merge with systolic size of 2 GPEs per merge and 4 GPEs per merge were compared against a regular sorting list merge, which is equivalent of systolic merge with only 1 GPE per merge. Static multiply allocation was used for all merge algorithms, and the workload tested were uniform random square matrices of $N = 1$K to 10K, with density ranging from $r = 0.0001$ to 0.01. The sorting list merge outperforms both systolic merge configurations for all matrices.

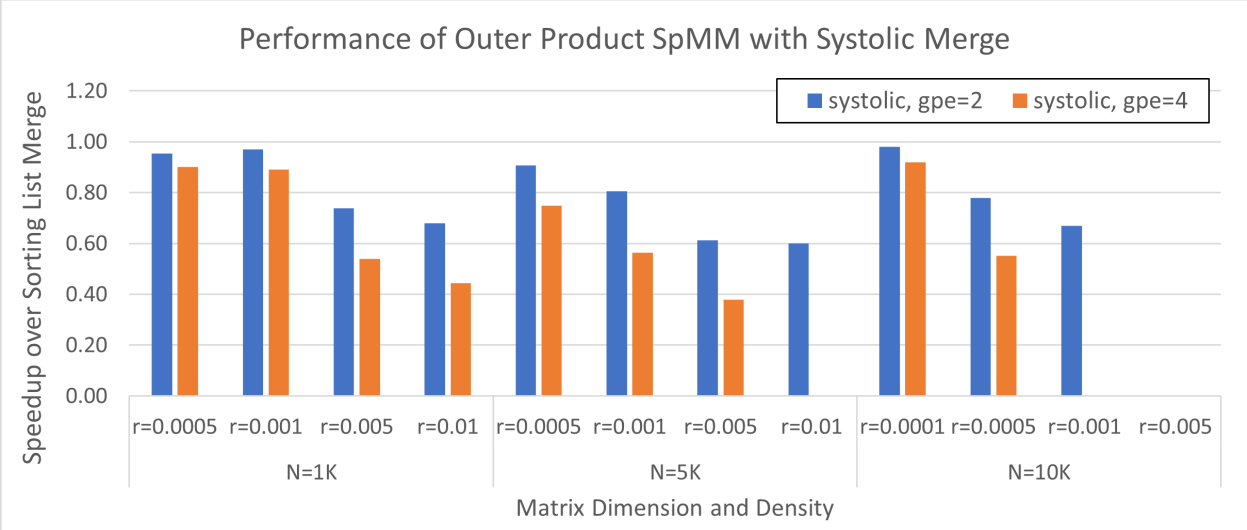


Figure 3.12: Performance of outer-product based sparse matrix-matrix multiplication with systolic merge against sorting list merge. Systolic merge with systolic size of 2 GPEs per merge and 4 GPEs per merge were compared against sorting list merge, which is equivalent to systolic size of 1 GPE per merge. Systolic merge performs poorly at all matrix size and density.

3.2.3 Rowwise Multiplication

While the outer-product method is the sparse matrix-matrix multiplication algorithm that results in the least amount of memory access by only fetching each nonzero element once, there is significant overhead in maintaining the large partial-product matrices and merging them together to produce the final matrix. Since outer-product method multiplies the columns of matrix A with rows of matrix B, every row in the final matrix C is subject to being merged until all the partial product matrices have been computed, making it difficult to split the workload into smaller partitions. The rowwise method resolves this problem by focusing on computing one row of final matrix before moving onto the next [66].

In the rowwise method, each row of the matrix A are multiplied with the row of matrix B that corresponds to the index of the nonzero elements of the given row [26], resulting in a row of the result matrix C as shown in Figure 3.13. Given a row of matrix A, each nonzero element in the row accesses the row of matrix C based its column index and perform a scalar vector multiply to produce the partial-product row. All the partial-product rows of a row of matrix A accumulate into a single row of matrix C corresponding to the same row position of the row of matrix A. Since the

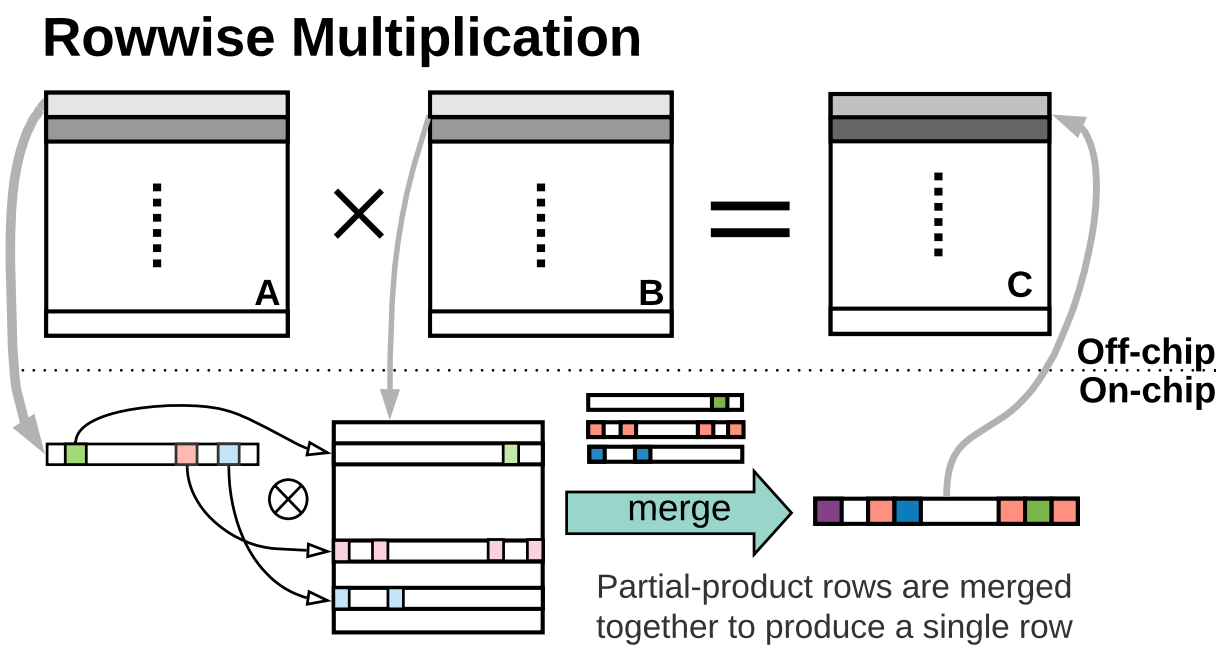


Figure 3.13: Overview of the Rowwise sparse matrix-matrix multiplication. For each row of matrix A, the nonzero elements are multiplied against the corresponding columns of B to produce the partial-product rows that merge together to a row of matrix C.

rows of matrix B are fetched again for each row of matrix A and the nonzero elements of matrix A can have overlapping column indices, it is quite likely for rows of matrix B to be accessed multiple times throughout the computation.

Similar to the outer-product method, the rowwise method does not require any index matching during the multiplication step. In addition, the rowwise method incurs less memory overhead compared to the outer-product method, because each row of matrix C is computed to completion before proceeding to the next row, instead of the entire matrix being computed at once. This allows only a single row worth of partial-product rows to be maintained in memory prior to the merge, which is a significant reduction from the outer-product method that has to maintain several partial-product matrices. However, the rowwise method requires the contents of matrix B to be accessed multiple times throughout its execution, thus it is important to configure to cache to take advantage of memory locality and data reuse.

3.2.3.1 Algorithm Mapping

With the rowwise approach, work scheduled from each row of matrix A results in a row of matrix C, allowing each task to be self-contained. Memory partitioning for the partial product rows is also simpler since all the partial-product rows generated from a row of matrix A corresponds to a single row of matrix C. In contrast to the outer-product approach that had to wait for all the partial-product matrices to be computed before proceeding to the merge phase, the rowwise approach can start performing the merge immediately after each row computation.

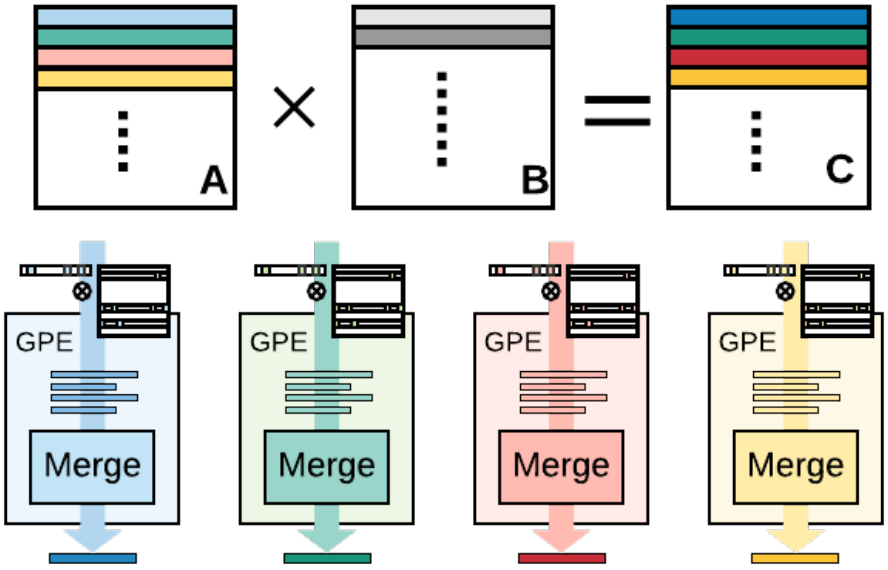


Figure 3.14: Work Allocation of Rowwise SpMM on Transmuter. Each GPE is assigned a row of matrix A that produces a row of Matrix C.

The rows of matrix A are distributed evenly across the tiles of the Transmuter, and each tile splits those rows amongst their GPEs, as shown in Figure 3.14. Each GPE fetches the nonzero elements of the rows of matrix A, and the corresponding rows of matrix B. While the nonzero elements of matrix A are fetched only once, the rows of matrix B can be fetched multiple times throughout the computation. In the worse case for a fully dense matrix, each element of matrix B gets N times, where N is the number of rows of matrix A. Each GPE performs the rowwise multiplication to generate the partial-product rows, then performs the merge to produce the corresponding row of

matrix C. The final row is written to memory, and the GPE proceeds to the next row.

3.2.3.2 Fixed Work Distribution

In rowwise multiplication, the multiplication phase and the merge phase happen at the same time by the same processing units. Therefore, the work distribution of the multiplication determine the load balancing of the merge phase. There are two potential ways for work to be distributed: fixed and variable.

In fixed distribution, each row of matrix C is assigned a fixed number of processing units, and the workload is always distributed evenly among the same set of processing units. The number of processing units assigned is based on the size of the partial-product rows that need to be merged, which is a factor of the size and density of matrix A and matrix B. The larger and denser the matrices are, there are more partial-product rows to be merged, so it is more efficient to spread the workload across larger number of processing elements. However, too many processing units assigned to a single row leads to higher overhead and wasted compute. Thus, it is important to profile the matrices beforehand to ensure the optimal distribution count.

3.2.3.3 Dynamic Work Distribution

While fixed distribution works well for matrices that have uniform distribution, it performs poorly on irregular matrices such as power-law graphs where only a portion of the matrix is, while the rest of the matrix is sparse. With these matrices, only a handful of rows will produce a large number of partial-product rows, while the rest of the rows will produce very little. Thus, a variable work distribution that optimizes for both extremes is preferable. In the variable work distribution, the number of processing units assigned to each row is based on the number of nonzero elements in the corresponding row of matrix A.

Performance of the rowwise algorithm with dynamic work distribution is compared against the static work distribution is shown in Figure 3.15. The dynamic allocation consistently outperforms the static allocation in most matrix size and density ranges swept by 2-16%, with the difference

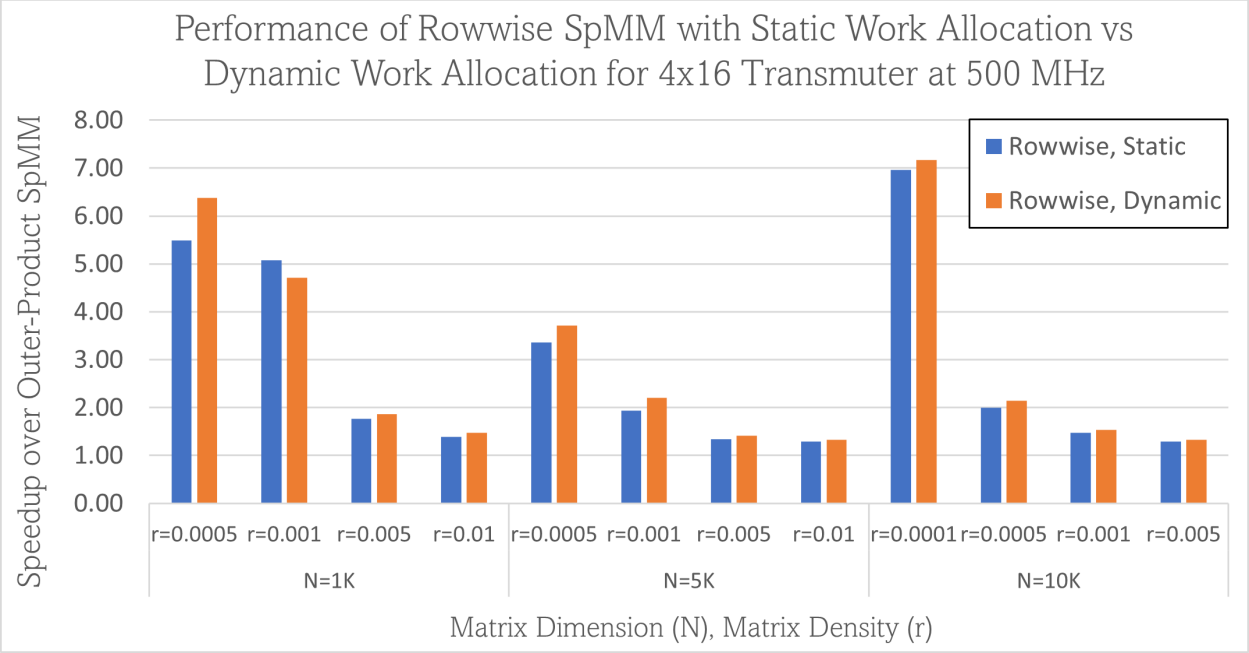


Figure 3.15: Performance comparison between rowwise SpMM algorithm with static work allocation and dynamic work allocation. Matrices are uniform random square matrices. Transmuter is configured to 4x16 tiles, with a 500 MHz clock and 4KB L1 private cache and 4KB L2 private cache.

diminishing for larger and denser matrices. Thus, dynamic work distribution is applied for all proceeding instances of rowwise algorithm unless specified otherwise.

3.2.3.4 Merge Algorithm

The core merge algorithm for the rowwise multiplication is the same as the outer-product method, where partial-product rows are accumulated together to form a row of matrix C. Unlike the outer-product merge, rowwise algorithm computes all the partial-products corresponding to a single row of matrix C from a single row of matrix A, so the merge step do not have to wait for other rows to be processed, and can be integrated into the multiply.

As discussed in the outer-product section, there are three different potential solution for the merge phase as shown in Figure 3.16: dense vector, sorting list, and systolic. For small matrices, the dense vector merge is the best solution thanks to its simplicity and low overhead. Sorting list merge incurs higher overhead than the dense vector due to the large sorting list structure, but the

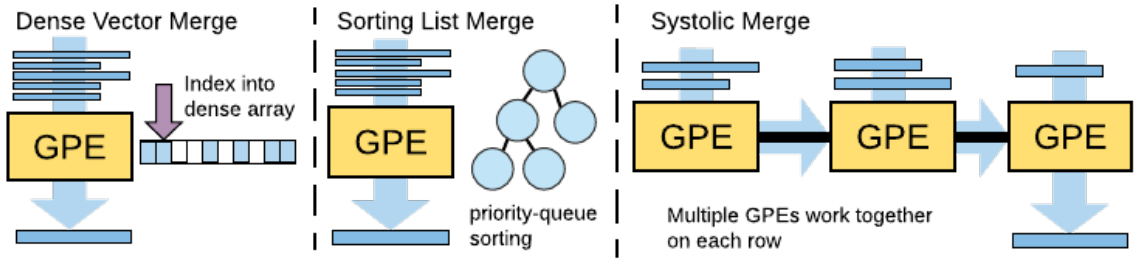


Figure 3.16: Three merge algorithms: dense vector, sorting list, and systolic.

algorithm scales better with larger matrices. Systolic method allows for the sorting list workload to be distributed across multiple GPEs. Because the multiply and merge are integrated together in the rowwise algorithm, the systolic method also allows the multiplication from a single row of matrix A to be distributed across multiple GPEs, leading to better load balancing in some situations.

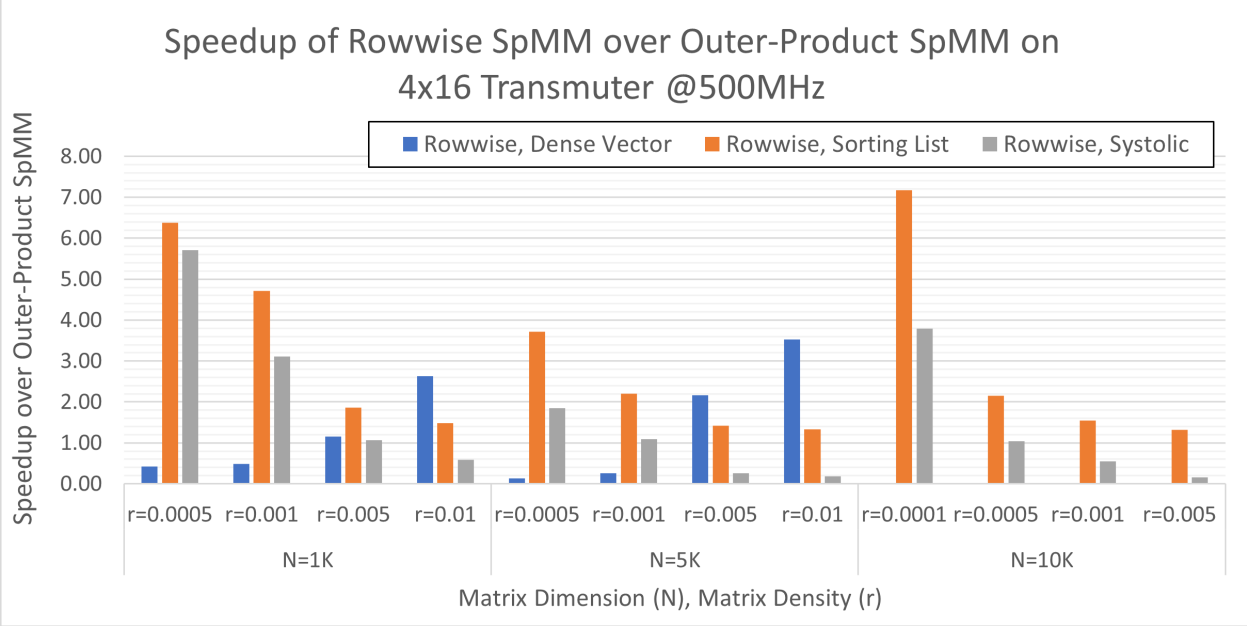


Figure 3.17: Performance of Rowwise SpMM against Outer-Product based SpMM on 4x16 Transmuter with 500 MHz clock. Three different merge schemes are compared: dense vector, sorting list, and systolic. Transmuter with 4KB L1 cache and 4KB L2 cache is configured to shared cache mode for outer-product SpMM, and private cache mode for all three Rowwise schemes.

Figure 3.17 shows the performance of the three different merge schemes of rowwise SpMM against the outer-product approach with sorting list merge. The matrices were uniform random

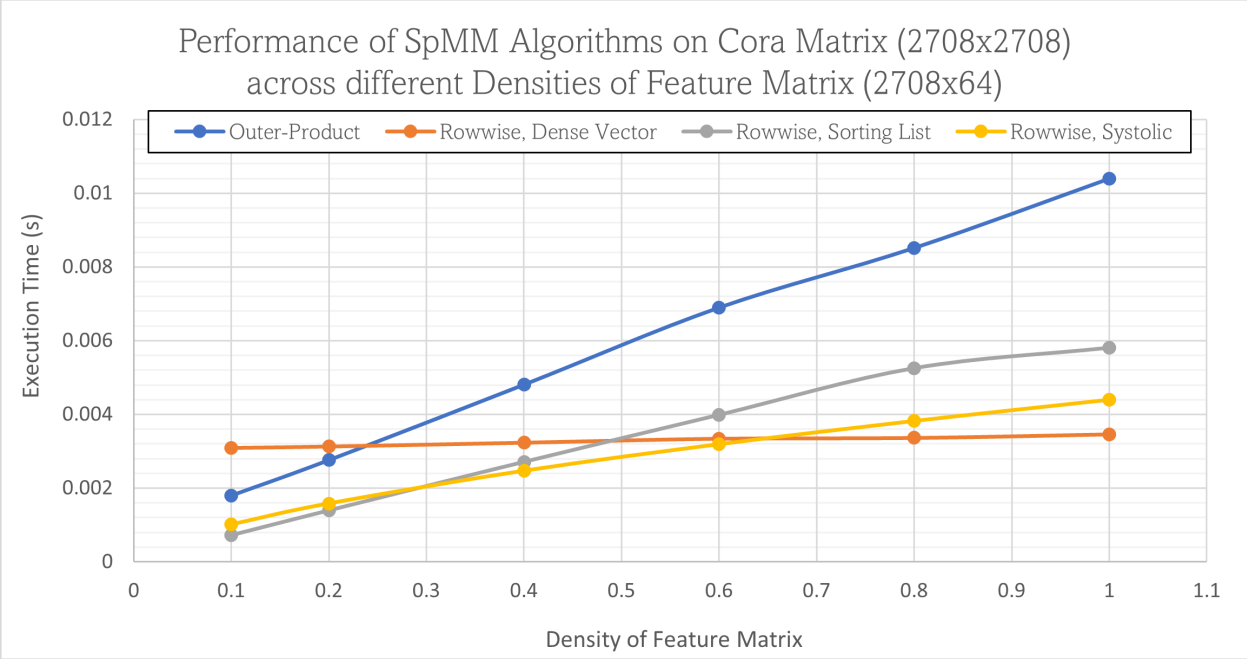


Figure 3.18: Performance of SpMM algorithms on Cora adjacency matrix across different feature matrix densities. Cora adjacency matrix (2708x2708) is multiplied against uniform random feature matrix of size: 2708x64. Transmuter is configured to 4x16 tiles, with a 1GHz clock and 4KB L1 cache and 4KB L2 cache.

square matrices with dimension N, and density r. Sorting list merge consistently outperformed the two other merge schemes especially as matrices got larger and sparser, with a speedup of up to 7.2x over the outer-product algorithm. When the matrix is denser than 1%, the dense vector merge starts to outperform the sorting list. However, when the matrix is too large ($N > 10K$), dense vector scheme drops in performance dramatically due to the high memory overhead of maintaining a dense vector array of size N. For matrices of these sparsity, systolic mode fails to perform better than the sorting list scheme.

3.2.3.5 Graph Convolutional Networks (GCNs)

Graph Convolutional Networks (GCNs) are an important deep learning workload for analyzing graph data that is widely used in many applications rangning from image classification, speech recognition, to natural language processing. The forward propagation of multi-layer spectral GCN is shown in Equation 3.1 [75]:

$$X_{l+1} = \sigma(AX_lW_l) \tag{3.1}$$

A is the adjacency matrix of the target graph, X_l is the input feature matrix for layer-l, W_l is the weight matrix of layer-l, and σ is the non-linear activation function. The adjacency matrix A is highly sparse with density less than 0.3%. The feature matrix X^l also starts with high sparsity of greater than 90% in its first layer, but becomes progressively denser at deeper layers due to weight matrix W being a fully dense matrix [21]. Thus, AX_l is a sparse matrix-matrix multiplication with a highly sparse adjacency matrix A and a rectangular feature matrix X with a narrow width and increasing density.

The AX_l portion of GCN kernel was tested by taking the CORA citation dataset [49], which is a 2708-by-2708 graph of academic publications and their citations, as the adjacency matrix A as shown in Figure 3.18. For the feature matrix X, a 2708-by-64 matrix was generated with density varying from 10% to 100% to reflect the increasing density of the feature matrix at deeper layers of the GCN. Rowwise approach with sorting list merge exhibits the best performance for densities less than 30%, but systolic merge starts to outperform the sorting list for the feature matrix density higher than 30%. This is due to the higher density of the feature matrix leading to more merging of partial products that share the same column index, which gets distributed across the systolic pipeline for parallel execution. Dense vector merge exhibits a flat execution time across all the densities, and starts outperforming the systolic merge for feature matrices denser than 65%.

3.2.3.6 Matrix Reordering

Matrix reordering algorithms reconstruct sparse matrices by altering the organization of its nonzero elements to aid in future processing or visual representation. The revere Cuthill-McKee algorithm (RCM) produces a matrix will less bandwidth by concentrating the nonzero elements towards the diagonal [11]. The rabbit ordering also reduces the bandwidth of the matrix, but also orders

the nonzero elements into more smaller local clusters [3]. The results of applying the RCM and rabbit reordering algorithms on the Cora citation database matrix is shown in Figure 3.19. The original Cora matrix has most of its nonzero elements evenly spread out, while the RCM algorithm concentrates the nonzero elements in small section along the diagonal. While the rabbit ordering still has high concentration of nonzero elements in the center, there are still some local clusters on the other regions of the matrix.

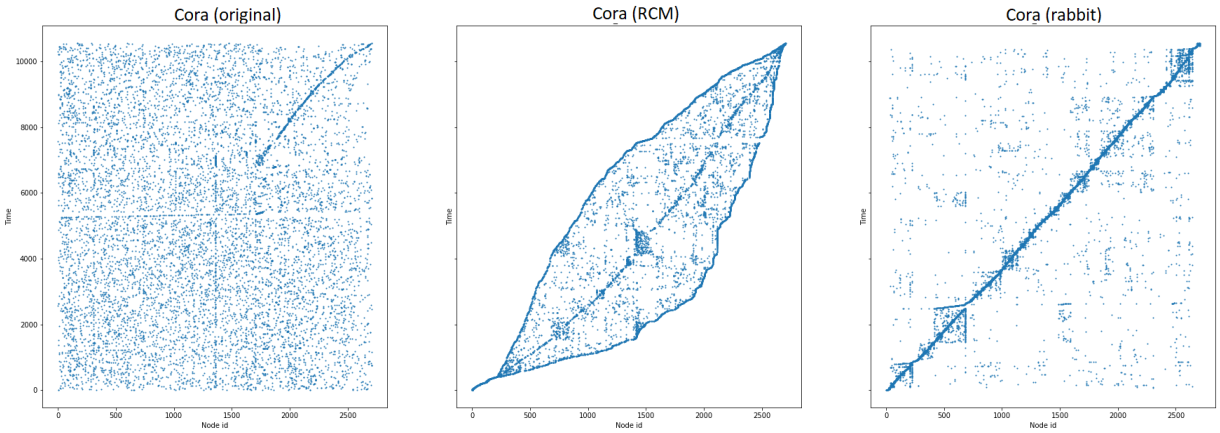


Figure 3.19: Visualization of the CORA citation database matrix, and the results of matrix re-ordering algorithms. Original Cora matrix is on the left. Result of applying reverse Cuthill-McKee algorithm is in the center, and the matrix on the right is the result of rabbit re-ordering.

The result of applying the RCM and rabbit matrix reordering on the Cora adjacency matrix is shown in Figure 3.19. For both dense vector and sorting list merge of row-wise SpMM, the matrix reordering had no noticeable impact on the performance of the algorithm. However, systolic merge showed noticeable change in performance, especially for the rabbit re-ordering, where the overall performance of the algorithm improved by 11-14%. Matrix reordering favors systolic merge because systolic merge can take advantage of the local clusters by splitting the work in those denser regions among multiple GPEs to have better load balancing. Overall, the rabbit order expands the region in which systolic merge performs the best from 30-65% to 20-70%.

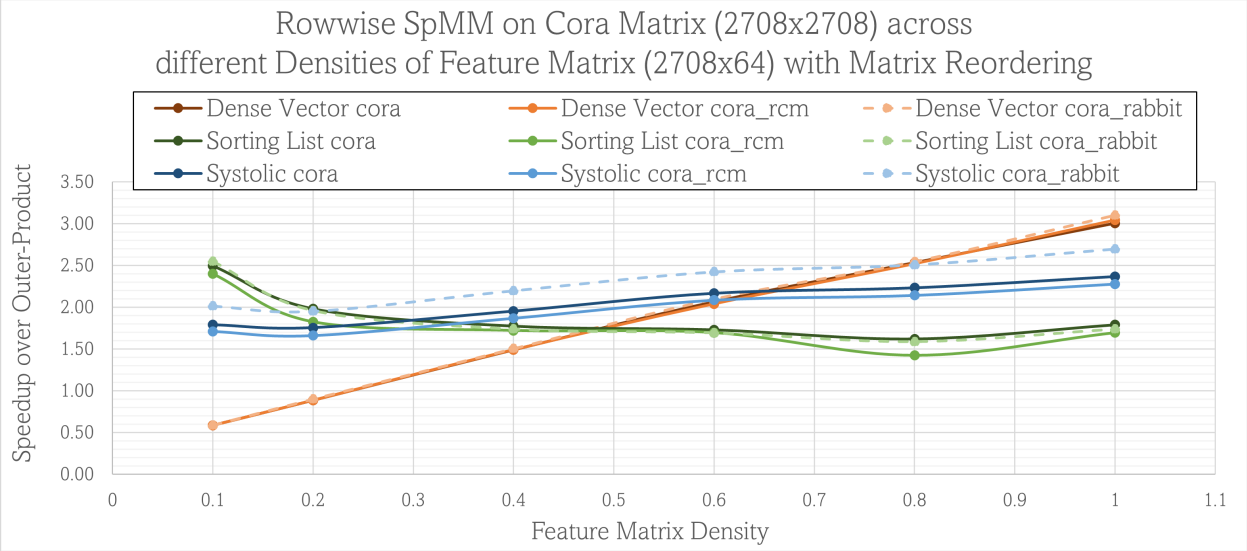


Figure 3.20: Performance of different Row-wise SpMM algorithms on Cora adjacency matrix with feature matrix re-ordering. Cora adjacency matrix (2708x2708) is multiplied against uniform random feature matrix (2708x64) with both reverse Cuthill-McKee (RCM) reordering and rabbit ordering. Transmuter is configured to 4x16 tiles, with a 1GHz clock and 4KB L1 cache and 4KB L2 cache.

3.3 Matrix-Vector Multiplication

Matrix-vector multiplication is one of the most widely important kernels in recent years, playing a key role in machine learning and image processing workloads. Similar to matrix-matrix multiplication, there is rising interest in matrix-vector multiplication for sparse data, as the input matrix grew bigger and bigger. Because matrix-vector operation can be treated as a subset of matrix-matrix multiplication, sparse matrix-vector workloads can be accelerated using similar techniques used in matrix-matrix multiplication.

3.3.1 Inner-product Algorithm

The inner-product matrix-vector multiplication is the standard algorithm commonly used for dense matrix inputs. Each row of matrix A performs a dot product with the input vector B to produce an element of the result vector. While the algorithm performs efficiently dense matrix and vector, when the input data are sparse the nonzero elements of the row of matrix A needs to be matched

with the nonzero element of vector B with the same column index. This addition of index-matching operation results in increased overhead, which can be overwhelming for very large, highly sparse sets of data.

3.3.2 Outer-product Algorithm

The outer-product matrix-vector multiplication seeks to minimize the index-matching present in the inner-product algorithm. Each column of matrix A in column index i performs a scalar multiply with an element of vector B at position i. The resulting partial product vectors are then merged together to form the result vector. For this study, both the matrix and the vector operands are in sparse format, and the final output vector is also sparse. The operation is split two key phases: *multipy* and *merge* phases.

In the *multiply* phase, the non-zero elements of the sparse vector operand are each scheduled to a separate processing element. Each assigned processing element then grabs the corresponding column of the spares matrix, and performs a simple multiplication to generate the partial product row. Because the outer product method fetches only the matrix columns that corresponds to the non-zero elements of the vector, there is no wasted memory loads and the workload scales linearly with the sparsity of the vector.

In the *merge* phase, the partial product rows generated by the *multiply* phase has to be merged into a single sparse vector. However, unlike the *merge* phase of SpMM where the merging of the partial product matrices was done in parallel by assigning a PE to each partial product row, there is only one partial product row that needs to be merged. Thus, the merging of the partial product row needs to be parallelized. While the ideal work distribution would be to assign a set index range for each PE to operate on, there is no way to determine the index distribution of the partial rows generated at the multiply. Instead, we propose a merge tree algorithm that performs the merge execution in parallel, as shown in Figure 3.21.

The merge tree is organized as a binary search tree, each node of the tree representing a GPE. The GPEs at the leaf are assigned a group of partial product rows to fetch from. These GPEs fetch

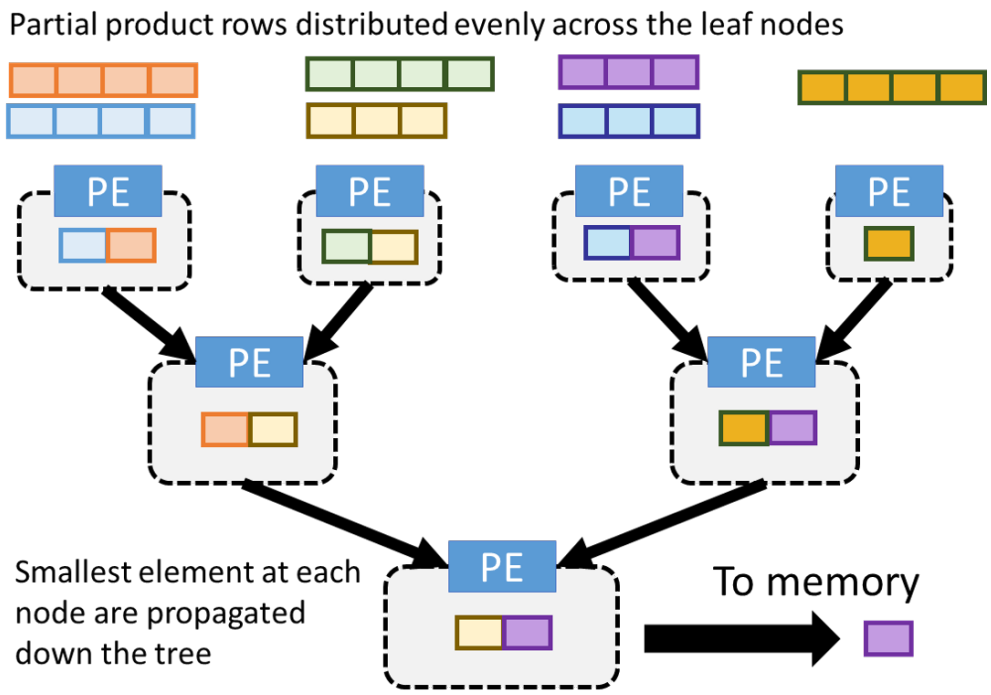


Figure 3.21: Merge Tree for Outer Product-based Sparse Matrix-Vector multiplication

and sort their assigned partial product rows in the same manner as the *merge* phase of SpMM; grabbing the leading element of each partial row, inserting them into a sorting list, and pushing out the current smallest element. The output element of these GPEs are propagated downward to their parent nodes, where the incoming elements are sorted again by each parent PE. The partial products propagate through the tree until they reach the root, where its output will be the current smallest element and written to memory. The merge tree fetches data from the partial product rows in parallel, and the sorting is conducted in pipeline fashion as each element propagates through the tree.

The performance of SpMV on Transmuter with 2x8 configuration is shown in Figure 3.22. The CPU baseline is the Intel i7-6700K with 4 cores at 4.0-4.2 GHz and 16 GB DDR3 memory, running MKL library. The GPU baseline is NVIDIA Tesla V100 with 5120 CUDA cores at 1.25 GHz and 16 GB HBM2 memory, running cuSPARSE library. The outer-product based SpMV algorithm with tree merging was run on Transmuter in two different modes: shared cache mode and private scratchpad mode. Shared cache mode and scratchpad mode exhibited similar performance

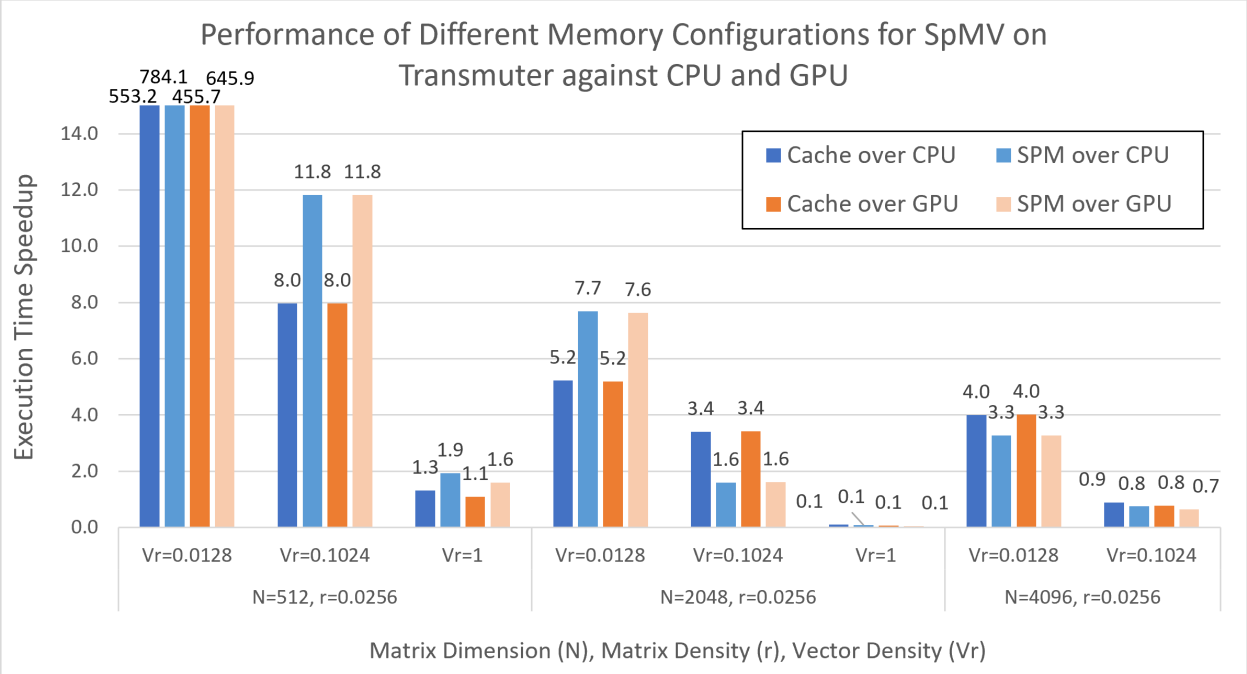


Figure 3.22: Performance of Sparse Matrix-Vector multiplication on 2x8 Transmuter for two different memory configurations over CPU and GPU. Cache mode is a L1 shared cache and L2 shared cache. SPM mode is L1 private scratchpad and L2 shared cache.

to each other in most of the workloads tested, with scrachpad mode outperforming cache mode by upto 20% when matrix dimension is low (N=512). Overall, both cache and scratchpad modes significantly outperformed the CPU and GPU implementations when density of the vector was less than 1, with the speedup ranging from 3.3-11.8x. Transmuter performs better at smaller matrices and higher vector sparsity, peaking at 553.2x and 784,1x over CPU and GPU respectively when N=512 and vector density is 0.0128. However, CPU and GPU exhibits better performance than the Transmuter when the vector is dense and the size of the workload is larger.

CHAPTER 4

Fast-Fourier Transform

4.1 Motivation

Signal processing algorithms are an integral part of low-power computing, especially on embedded systems. Fast-Fourier Transforms (FFT) in particular has been a fundamental building block for signal interpretation and system analysis over several decades [4, 57]. The rise of smartphones and Internet-of-Things (IoT) have shifted the signal processing efforts towards embedded, low-power platforms that do not fit the Transmuter. However, it is still important for Transmuter to achieve high performance efficiency on FFT, as it is still a critical kernel for a wide range of scientific computing workloads.

4.1.1 Contributions

The initial development and analysis of the systolic FFT code using the 1D-systolic mode of the Transmuter was done by Sung Kim [53]. My contributions were the addition of the register-to-register version of the systolic FFT and the subsequent analysis of its performance.

4.2 Fast-Fourier Transform

Fast-Fourier Transform (FFT) is a signal processing algorithm that computes the discrete Fourier transform (DFT) of a signal in rapid fashion. The discrete Fourier transform is defined by the

following equation:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk}, \text{where } k = 0 \text{ to } N-1 \tag{4.1}$$

DFT takes a signal in time domain and converts it to a frequency domain representation, which helps analyze the different frequency components of the signal. FFT is one of the fundamental building blocks of signal analysis, and is a core component of many signal processing workloads. There has been extensive work over the decades in accelerating FFT on various different platforms, ranging from traditional architectures such as CPUs and GPUs [50, 9], to FPGAs [61], as well as specialized ASICs [30, 43]. With a reconfigurable architecture such as the Transmuter, the hardware can be scaled to match the input size of the workload [53] . The register-to-register functionality of Transmuter also allow the traditional systolic array implementation of the FFT to be performed efficiently with a chain of reconfigurable cores.

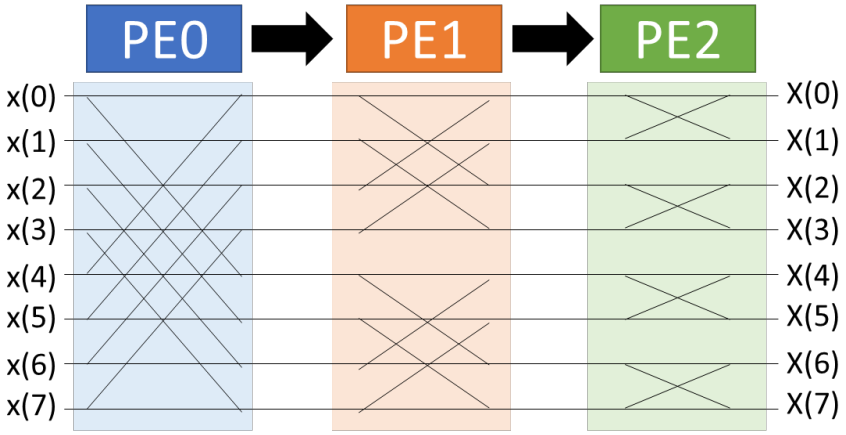


Figure 4.1: Diagram of systolic array-based FFT with N=8, mapped onto the Transmuter system.

4.2.1 Systolic Array-based Algorithm

In low power, embedded systems where energy efficiency is a critical factor, FPGA and ASIC solutions are widely used to accelerate the FFT algorithm. Many of the FFT algorithms on these

platforms employ a systolic array design, where a pipeline of processing units compute each stage of the fourier transform. The systolic array based FFT solutions achieve high energy efficiency thanks to the low communication overhead between each stages of a systolic array. By transferring the intermediary results through a direct connection that is consumed immediately by the subsequent stages, the systolic array eliminates the need to allocate intermediary storage, keeping the hardware and energy footprint low.

Figure 4.1 shows how the systolic array FFT of N=8 is mapped onto a Transmuter system. The first GPE receives the input signal from the LCP and computes the first stage. The results produced by the GPE is sent to the adjacent GPE as they are produced in pipeline fashion. The intermediary GPEs receives the incoming signal from the previous GPE, and sends its own results to the next GPE directly. At the last GPE, the final computation is performed and the result is written directly into memory.

4.2.2 Mapping on Transmuter

For an FFT of size N, there are only $log_2(N)$ stages to be computed and allocated across the system. Therefore, only $log_2(N)$ GPEs are set to active, and the remaining tiles, GPEs, as well as the cache banks are power-gated. Two different mechanisms for inter-GPE communication were tested for the Transmuter: systolic array configuration using scratchpad-based FIFO queues, and a register-to-register pipeline.

In systolic array configuration, a hardware FIFO queue transmits messages between the adjacent GPEs. The L1 cache bank is configured to private scratchpad memory, and parts of the scratchpad memory is partitioned as queues for the FIFO. Each FIFO queues form direct, one-way communication between an adjacent GPE similar to the work queues between a LCP and a GPE. The FIFO queues are accessed by each GPE through a memory-mapped address and are called using a dedicated API.

With the register-to-register communication, a handful of floating-point registers are partitioned to host a dedicated FIFO between the GPEs. Unlike the systolic mode where the L1 cache bank has

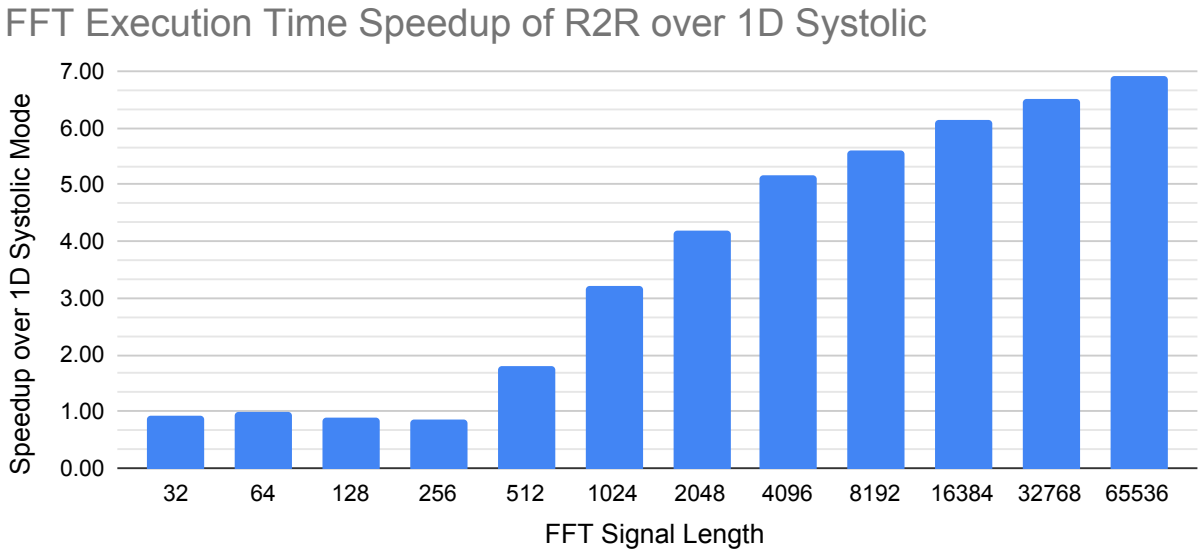


Figure 4.2: Execution time speedup of FFT using register-to-register communication over 1D systolic mode. Register-to-register exhibits speedup of upto 6.90x over the 1D systolic mode implementation.

to configured to private scratchpad, the register-to-register communication can be used on private or shared cache modes. Thanks to the register-to-register connection, the GPEs, which consists of low power, off-the-shelf general-purpose cores, can adapt to a systolic paradigm without the need of expensive hardware barriers or message passing mechanisms. The register-to-register takes advantage of the existing control mechanisms of the architecture pipeline, and re-purpose specific registers from the core's ISA to minimize the need of specialised APIs.

Figure 4.2 shows the performance of FFT on Transmuter using systolic mode against register-to-register communication. Two modes perform similarly at signal lengths $N < 512$, when the amount of data transferred between the GPEs are low. But as the signals get longer, the register-to-register communication exhibits better and peaks at $N = 64$k with 6.90x speedup.

This is further reflected on the comparison of energy efficiency of the two implementation as shown in Figure 4.3. Register-to-register exhibits better energy efficiency than systolic mode for all signal lengths, and systolic mode suffers dramatic reduction in performance for $N > 512$, where the efficiency drops to approximately 1 GFLOPS/W. Register-to-register keeps increasing in efficiency as signal length is increased, peaking at $N = 64$k with 9.31 GFLOPS/W. This is

due to a $N = 64$k FFT requiring 16 GPEs, which utilizes all the GPEs within a 1x16 tile of the Transmuter. For $N < 64$k, some GPEs need to be power-gated but there were still some static power lost from idle cross-bar connections or cache banks.

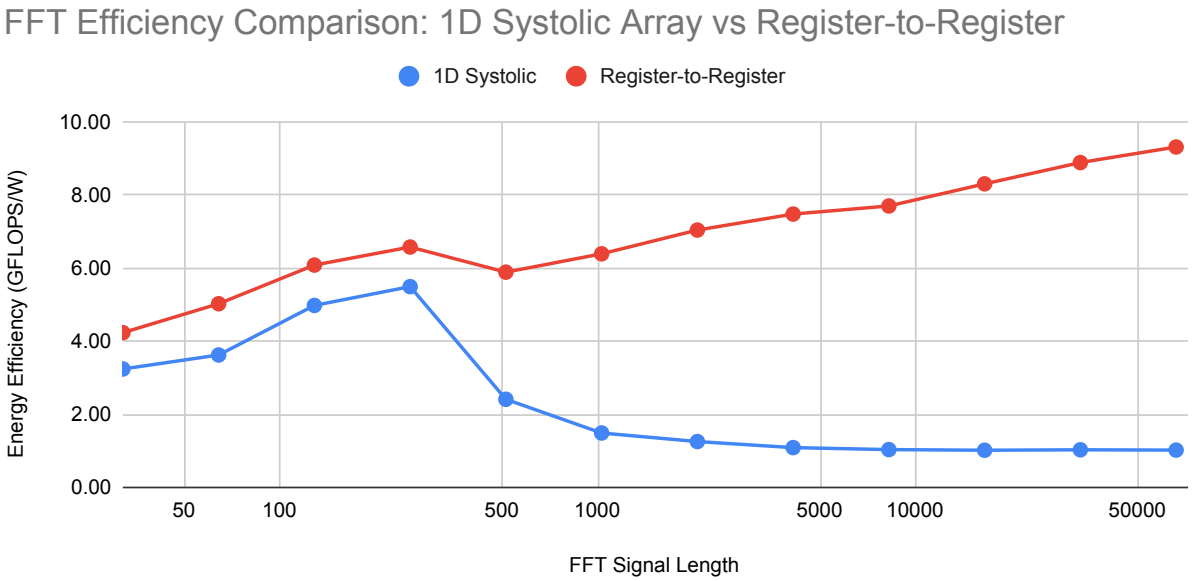


Figure 4.3: Energy efficiency of FFT for both 1D systolic mode and register-to-register at 1 GHz clock, with pre-computed twiddle factors. Systolic mode drops in efficiency for signals greater than $N = 256$, and stagnates at around 1 GFLOPS/W for larger signals. Register-to-register improves in efficiency as signal gets larger, and peaks at 9.31 GFLOPS/W for N=64K.

4.2.3 Twiddle Factor Computation

There are two different modes of operation for the FFT: pre-computed and on-the-fly. In the pre-computed mode, the twiddle factors of each FFT stage are calculated prior to the execution, and loaded from memory during execution. In the on-the-fly mode, the twiddle factors are calculated during execution of each stage. Pre-computing the twiddle factors lowers the computational load of the FFT execution, while increasing the memory overhead as each twiddle factor needs to be fetched from memory. Meanwhile, the on-the-fly mode requires the GPEs to perform complex arithmetic at each stage, while minimizing the burden on the memory.

Figure 4.4 shows the performance of on-the-fly computation of twiddle factors against loading

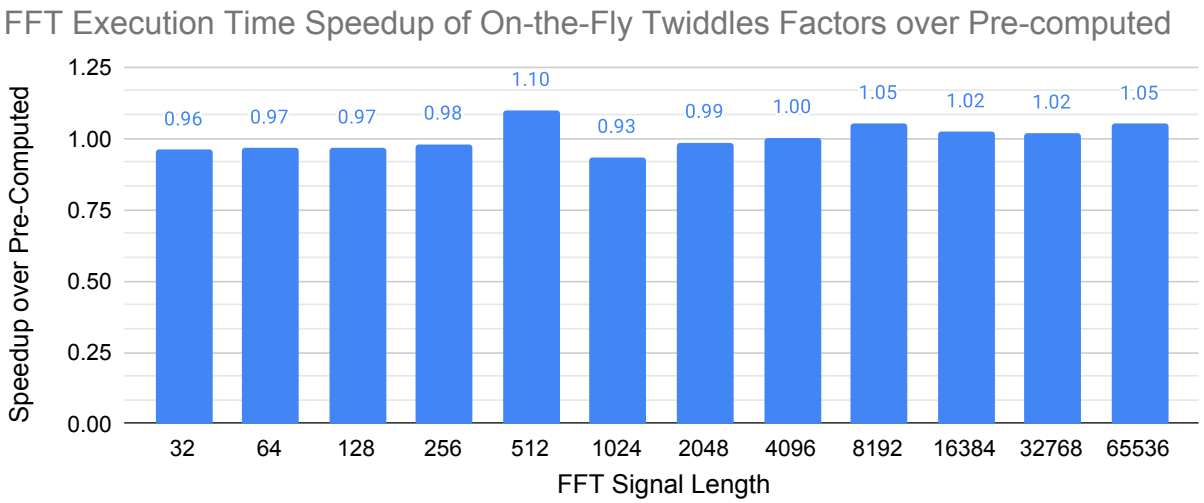


Figure 4.4: Execution time speedup of on-the-fly twiddle factor computation against pre-computed twiddle factors. Pre-computed mode has a slight edge for smaller signal sizes, but on-the-fly has better performance for longer signals

pre-computed twiddle factors. For smaller signals ($N < 512$), the pre-computed twiddle factors show a slight edge over the on-the-fly computation. However, the on-the-fly computation shows a speedup of upto 1.05x when the signal is longer. The pre-computed mode shows higher performance at lower FFT lengths, while the on-the-fly mode scales better as FFT size increases. This is because the memory overhead of loading the twiddle factors is mitigated by the cache for small FFTs, but once the twiddle factors are too large to fit properly in the cache, on-the-fly computation results in better performance.

4.2.4 Cache vs Scratchpad

One of the main bottlenecks of pre-computed mode is loading of the pre-computed twiddle factors. The scratchpad mode of the Transmuter allows the twiddle factors to be stored in a fast, scratchpad memory that will never be evicted out to memory. In addition to the twiddle factors, the scratchpad can also be used to store the intermediary signals before they are ready to be computed.

Three different memory configurations of Transmuter were explored for the FFT, on both pre-computed and on-the-fly twiddle factor computation: private cache, shared scratchpad, and hybrid

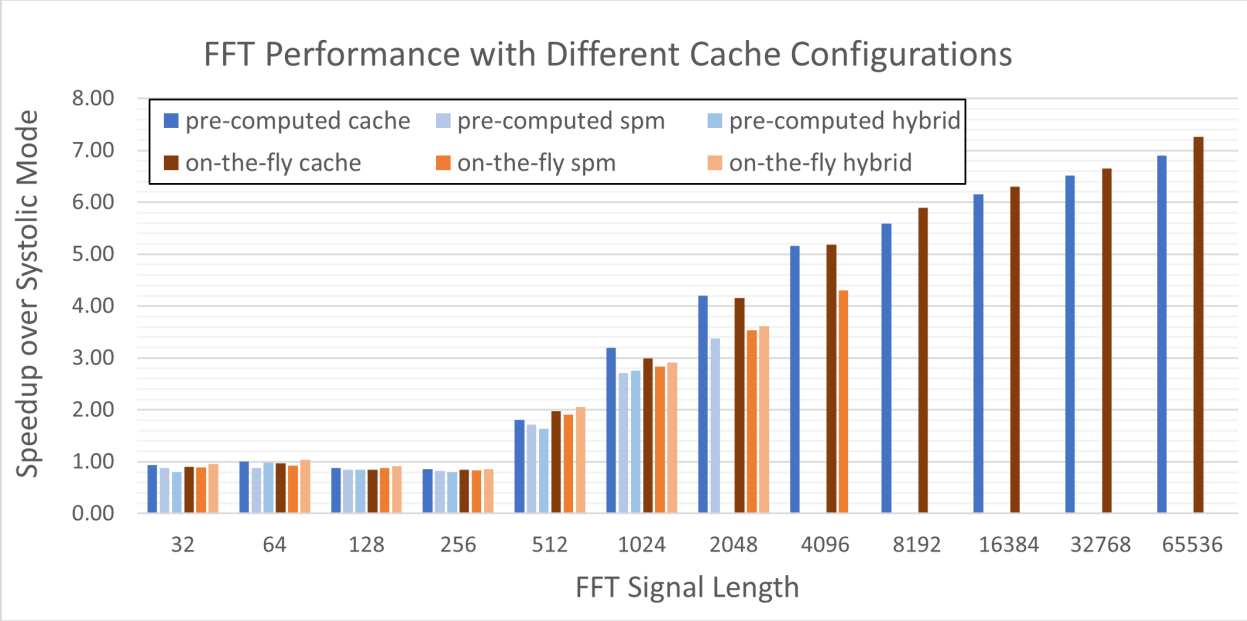


Figure 4.5: FFT Performance of different cache configurations on Transmuter. Speedup of the execution time over the systolic array mode was compared for private cache, shared scratchpad, and hybrid mode for each twiddle factor modes (pre-computed and on-the-fly)

mode. The private cache mode configures the L1 cache banks of each GPE to private cache, in order to maximize the memory locality. In the shared scratchpad mode, all the L1 cache banks of the GPEs are grouped together to form a uniform, shared scratchpad. The pre-computed twiddle factors and intermediary data are stored in this shared scratchpad space, and the GPEs only have the shared L2 cache as their lowest level cache. The hybrid mode allocates half of the L1 cache banks for a shared scratchpad while partitioning the rest of the cache banks to serve as a shared cache. The hybrid mode seeks to mitigate the negative impact of not having a proper cache while still giving access to a shared scratchpad.

Comparing the performance of each cache mode as shown in Figure 4.5, the private cache mode has the best performance for all FFT sizes for both pre-computed and on-the-fly computation, followed by the hybrid mode and scratchpad mode. The cost of loading the pre-computed twiddle factors from memory appears minimal compared to the penalty of a smaller local cache that negatively impacts the overall performance of the algorithm. While the hybrid mode appears to mitigate some of that downside, there is not enough benefit of maintaining a scratchpad for the purpose of

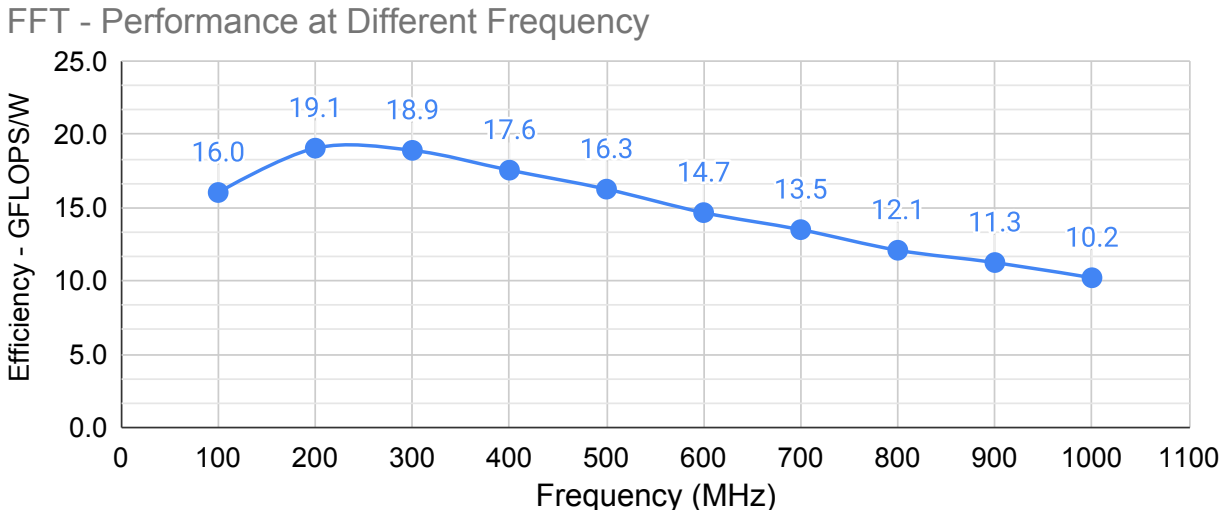


Figure 4.6: Efficiency of $N = 64$k FFT at different clock frequencies. Transmuter exhibits the best performance of 28.4 GFLOPS/W with a 200 MHz clock.

storing the twiddle factors and intermediary data. Because there is little memory overlap between the GPEs, private cache is the most optimal memory configuration for the FFT algorithms.

4.2.5 Optimizing for Energy Efficiency

The metric used for measuring energy efficiency is floating-point operations per second per watt (GFLOPS/W). When counting the floating-point operations (FLOP), the twiddle factor computations of the FFT needs to be removed from the count, as it is not part of the main FFT calculation and it is an unfair comparison for the pre-computed mode. So rather than obtain the value empirically, the FLOP count is calculated as $5N*log(N)$, which is the minimum floating-point operations for a signal of length N. This way, the FLOP count of FFT remains the same for each signal length N, regardless of the algorithm. The normalizing of the FLOP count allows GFLOPS/W to be depend solely on the execution time and the power consumption of the implementation.

To optimize for maximum energy efficiency, the on-the-fly FFT with $N = 64$k was executed at clock frequencies ranging from 100 MHz to 1 GHz as shown in Figure 4.6. Transmuter was configured to 1x16, L1 private cache and L2 private cache, with a register-to-register queue of depth 4. The performance peaks at 200 MHz with 19.1 GFLOPS/W, and steadily decrease with

increasing frequency.

CHAPTER 5

Deep Neural Networks

5.1 Motivation

Machine learning workloads, especially deep neural networks (DNNs), have been growing at a staggering rate over the past decade, and has been applied to many different areas in recent years. There has been an growing interest in applying DNNs on graph-based data, due to the ever growing size of graph datasets [79]. Graphsage is a neural network developed by Stanford to perform inductive representational learning on large graphs [27]. The Graphsage workload help showcase the reconfigurability of the Transmuter, as well as the hybrid scratchpad mode feature for performing dense matrix-matrix multiplication.

5.1.1 Contributions

The Graphsage workload was developed on Transmuter in collaboration with Agreen Ahmadi, who helped with the initial implementation of the forward pass. I lead the development of the hybrid mode optimization as well as the backpropagation kernel.

5.2 Algorithm Overview

Graphsage is a neural network model used on graph-structured data [27]. The network is used to predict the properties of a given node based on the known properties of its one-hop neighbors as

well as the two-hop neighbors derived from the one-hop neighbors. The entirety of Graphsage workload, from inference to training, can be broken down into four main phases: embedding, forward pass (or inference), weight update, and back propagation.

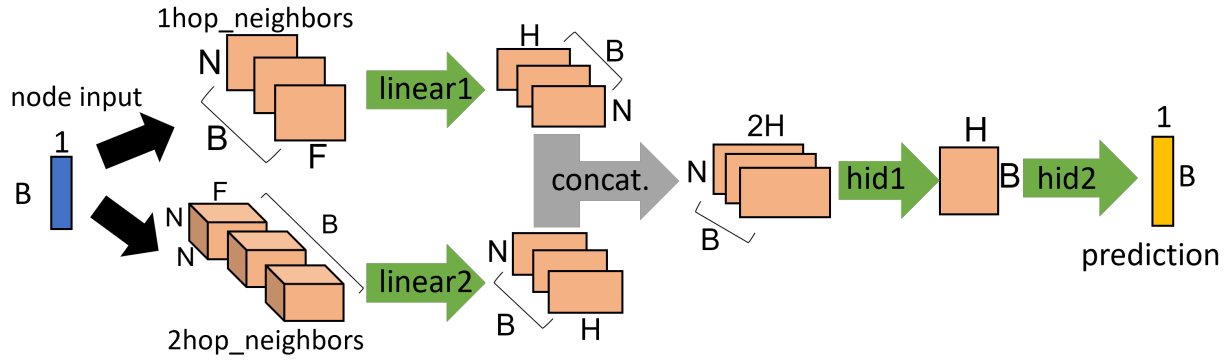


Figure 5.1: High-level diagram of the Graphsage neural network, showing the primary linear layers.

5.2.1 Graphsage Network Overview

The high-level diagram of the Graphsage network is shown in Figure 5.2. Graphsage network starts with a batch of target nodes to predict. The one-dimensional array of nodes is passed through a sampling layer where the 1-hop neighbors of each of node is selected. Each of these 1-hop neighbors are sampled again to form a group of 2-hop neighbors. These sets of 1-hop and 2-hop neighbors are passed through the embedding layer to extract the known properties of the neighbor nodes to create a 3D and 4D arrays, respectively.

Each set of arrays from the 1-hop and 2-hop neighbors are then passed through linear input layers with a rectified linear activation function (ReLU). The resulting 4D array is passed through a mean pooling layer that takes the average of the 2-hop neighbors to match the dimensions of the results from the 1-hop neighbors. The resulting set of 3D matrices derived from 1-hop and 2-hop neighbors are concatenated together to form a single 3D array matrix.

This internal matrix is passed through a linear hidden layer with ReLU activation. The resulting 3D array is further reduced down to a single 2D matrix through a second mean pooling layer that

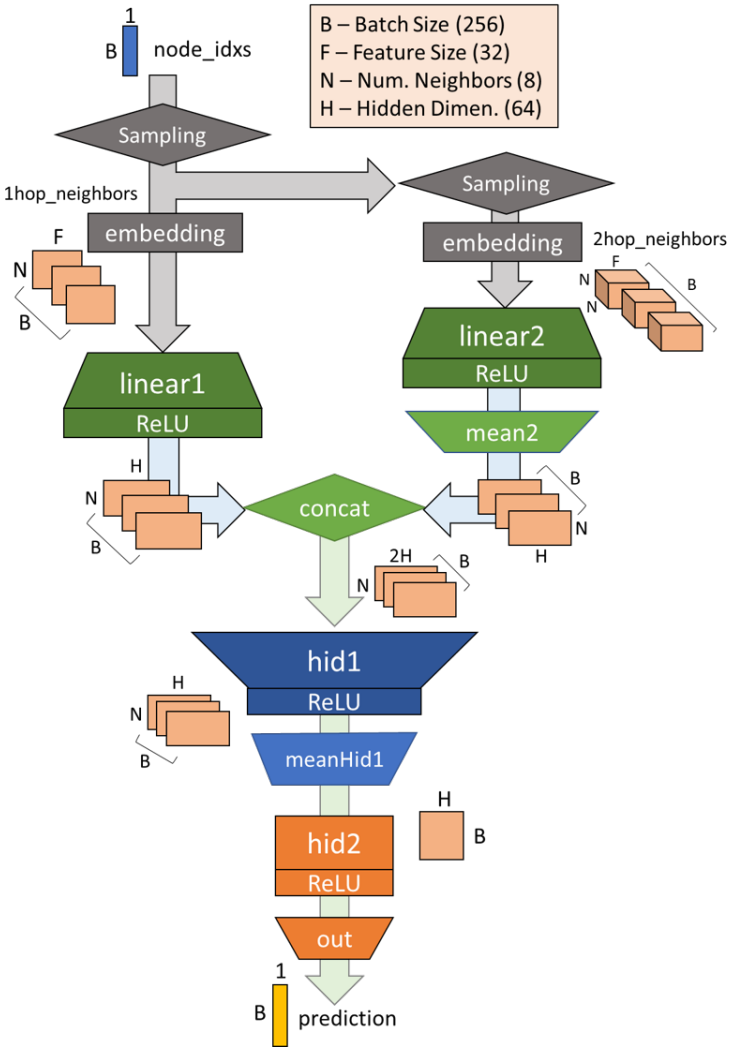


Figure 5.2: Detailed diagram of the Graphsage neural network.

takes the average of the values from the neighbors and reduce it to a single node. Finally, the 2D matrix that contain the hidden variables for each of the nodes in the batch is passed through the linear output layer with ReLU activation to produce the vector of predicted values for each of the nodes in the batch.

5.2.2 Sampling and Embedding

The sampling and embedding phase is a memory-driven kernel that sets up the input arrays that feed into the input layers of the Graphsage neural network. Given a target node, a sample of

N neighbors are selected at random. These are the 1-hop neighbors. Then a sample of 2-hop neighbors are chosen by taking a random sample of N neighbors from each of the 1-hop neighbors. Thus, the total number of nodes selected from the sampling from one target node is $N + N^2$. The embedding kernel extracts the known features of these neighbors from a look-up table to generate the input arrays. If the feature size is F, the size of the input arrays for the 1-hop and 2-hop neighbors are $N * F$ and $(N + N^2) * F$, respectively, for each target node. Because the kernel is dominated by nested memory accesses and pointer-chasing, the performance is primarily memory-bound, but can be parallelized across the sampled neighbors.

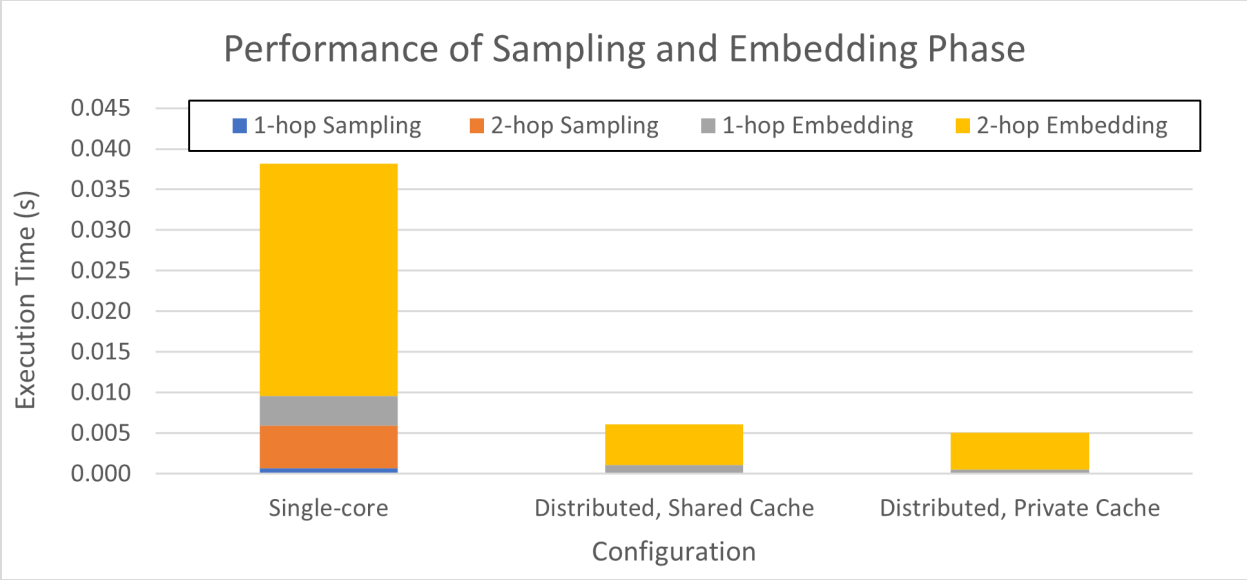


Figure 5.3: Performance comparison of the sampling and embedding phase for different configuration. Distributed workload in private cache configuration exhibits 7.5x speedup over serial implementation.

The neighbor sampling and embedding workload was mapped onto the Transmuter by distributing the batch of target nodes across all the GPEs of the Transmuter. Because the 1-hop neighbor sampling and embedding process is mostly read-only other than setting up the resulting feature array, the entire process can be spread across the different GPEs without concern for coherence. The memory space for the feature arrays is pre-allocated, and properly blocked for each GPE to prevent any unwanted write conflicts. For the 2-hop neighbor sampling, these second set of neighbors are selected based on the sampled 1-hop neighbor. Therefore, the work for 2-hop neighbors should not

be distributed until all the 1-hop neighbors are chosen to ensure coherency across different tiles.

As shown in Figure 5.3, the parallelization of the neighbor sampling and embedding resulted in a speedup of up to 7.5x for the private cache configuration. The breakdown of the kernels in Figure 5.4 shows the embedding of the 2-hop neighbors dominating the overall execution at 89%.

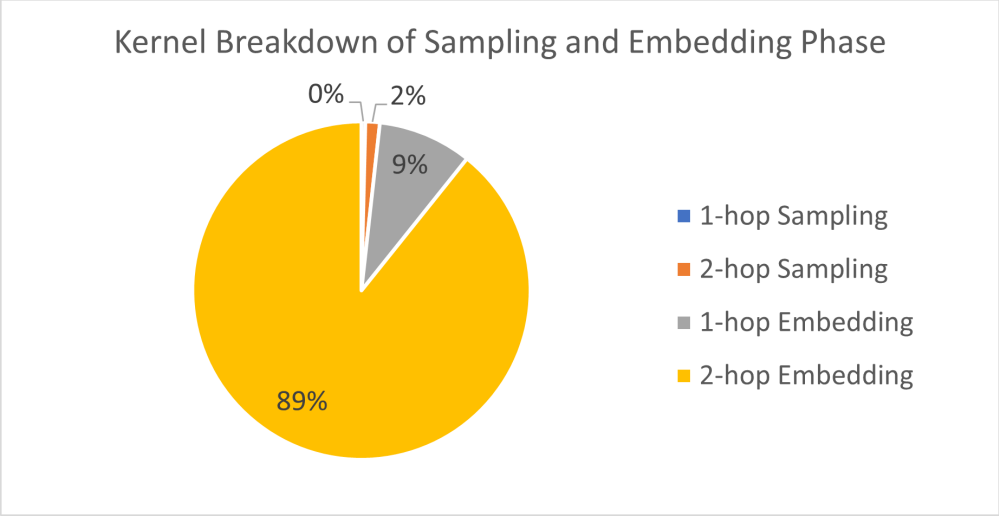


Figure 5.4: Kernel breakdown of the sampling and embedding phase. Majority of the execution resides in the embedding phase of the 2-hop neighbors.

5.2.3 Forward Pass

The forward pass, or inference, of the Graphsage network contains two input layers, two hidden layers and one output layer. They are all linear layers with ReLU activation function. There are also two mean pooling layers, as well as a concatenation step to combine the signals from 1-hop neighbors and 2-hop neighbors. As shown in the breakdown of the forward pass workload in Figure XX, majority of the computation lies in the processing of linear layers, which is a dense matrix-matrix multiplication kernel. In addition, to speedup the backpropagation portion of the neural network training, the output gradient calculation is done in tandem with the inference to minimize access to the memory. The output gradient calculation is, a nested, dense matrix-matrix multiplication that incorporate the inputs and outputs from the inference processing.

Two different mapping schemes was tested for the dense matrix-matrix multiplication of the

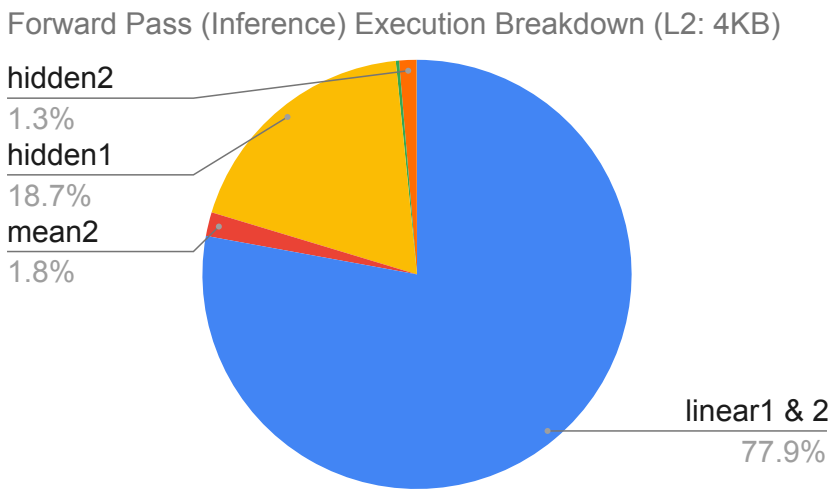


Figure 5.5: Breakdown of the forward pass (inference) kernel. The linear layers take up the most computation time, followed by the hidden layers.

linear layers: coarse and fine. The coarse partitioning is a naive work distribution where each node in the batch is assigned a single GPE. Each GPE processes the matrix-matrix multiplication pertaining to its batch on its own so that there is no coherency conflicts between the tiles or the individual GPEs. Because each GPE works independently of each other, there is no resource sharing between the GPEs, so the most optimal cache configuration is L1 private cache and L2 private cache.

In the fine-grain partitioning, the batches are distributed amongst the tiles, but the GPEs within each tile operate on the same batch together. The weight matrix of a linear layer is a shared constant amongst all the batches, and needs to be accessed multiple times as each tile operates on multiple batches, so the scratchpad can be utilized for the weight matrix of each linear layer.

Three different L1 cache modes were tested for the fine-grain partitioning: private cache, private scratchpad, and hybrid. The private cache mode is the baseline configuration, and is optimal for parts of the inference workload where the scratchpad is not utilized, such as the pooling layers. In private scratchpad mode, each GPE has its own scratchpad to operate on. While this mode maximizes the size of the available scratchpad, the lack of any L1 cache hinders the performance significantly. The hybrid scratchpad mode allows the GPEs access to a shared L1 scratchpad as

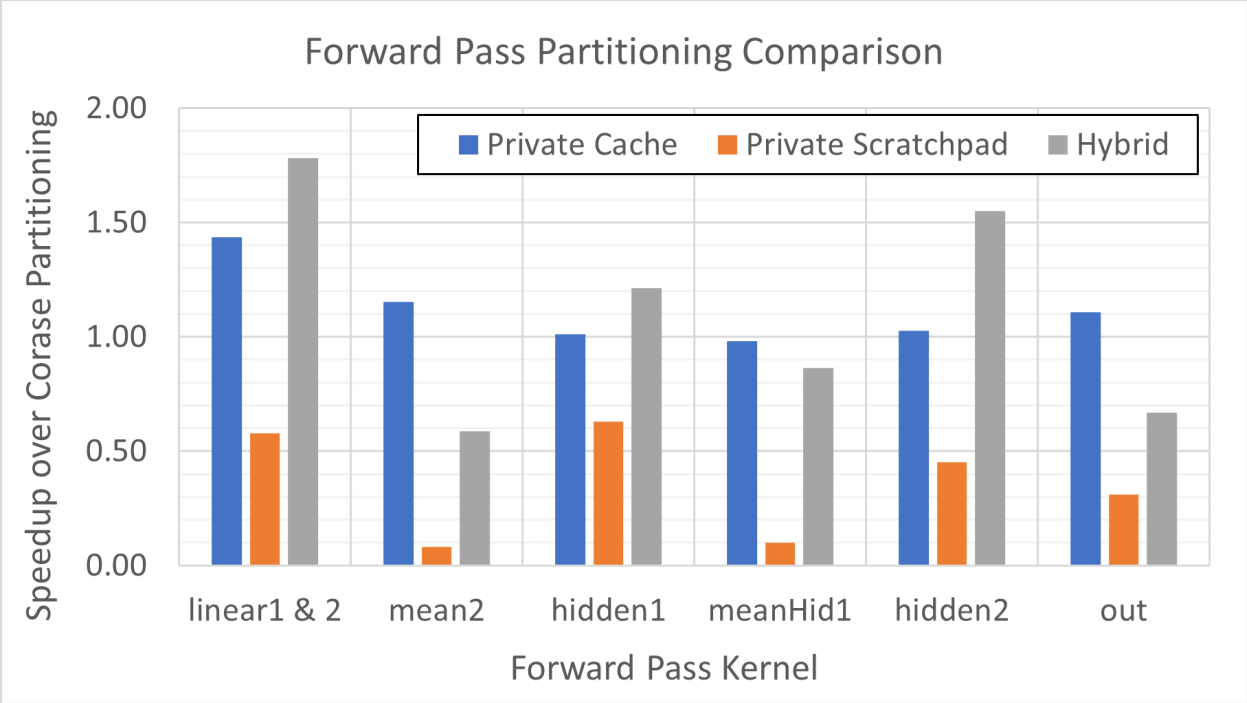


Figure 5.6: Speedup of fine-grain partitioning for different L1 cache modes. Hybrid cache mode is the most optimal for the linear and hidden layers, which private cache performs the best for the mean pooling layers.

well as a shared L1 cache. While the total scratchpad capacity available is lower than that of the scratchpad-only modes, for the 4KB L1 cache bank configuration, the scratchpad capacity was not an issue.

Figures 5.6 shows the performance of fine-grain partitioning at different L1 configuration over the performance of the coarse partitioning. For the linear and hidden layers that utilize the scratchpad, the hybrid mode outperforms the private cache and private scratchpad modes by up to 1.5x and 8.6x, respectively. Private scratchpad mode performed poorly for all the kernels, even against the coarse partitioning due to the lack of any L1 cache. As expected. the private cache mode performed the best for the pooling layers that did not utilize the scratchpad. Because there is a significant performance gap between the private cache mode and the hybrid mode in the kernels that they are the best at, dynamic reconfiguration that switches between the two modes during execution will lead to the most optimal performance.

5.2.4 Backpropagation

The output computation of the linear layer can be expressed in the form $Y = XW$, where Y is the output matrix, X is the input matrix, and W is the weight matrix. The gradients of the input and the weight can be derived by taking the derivative of the loss, L, which leads to the expressions:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} W^T \qquad\qquad \frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Y} \tag{5.1}$$

The gradients of the weight matrix of each of the linear layers are computed by propagating the input gradient backward to get the output gradient. Similar to the forward pass, the backpropagation workload is a series of dense matrix-matrix multiplication kernel involving the weight matrix. Therefore, a weight partitioning scheme similar to the forward pass is applied to the backpropagation, and the hybrid mode is used to store the weight matrix in the scratchpad. While it would be beneficial to store the input gradient in the scratchpad to be utilized in the subsequent layer as the output gradient, the size of the input and output signals are too large to fit in the limited capacity of the L1 shared scratchpad in the hybrid mode.

For the first two linear layers which are smaller in size compared to the hidden layers, the weight gradient was also stored in the remaining space of the scratchpad. This weight gradient was used to perform the weight update immediately following the backward pass of the two layers, to minimize additional memory access. In addition, the layout for the data structures of the gradient signals were switched from row-order to column-order, to better accommodate the data access patterns of the backpropagation calculations.

5.2.5 Weight Update

Once the backpropagation is complete, the weight gradients of each layer is used to calculate the new weight matrix. The equation for the new weight is as shown in Equation **??**, where W is the

weight matrix, a is the learning rate, and $\frac{\partial L}{\partial W}$ is the weight gradient from the backpropagation. c

$$W_{new} = W_{old} - a(\frac{\partial L}{\partial W}) \tag{5.2}$$

The weight update is a simple linear algebra kernel involving scalar multiplication and matrix subtraction. The workload is distributed across all the GPEs, with proper data blocking to ensure there is no cache conflict between the different cores. Because each core is self-contained and there is little data reuse between the GPEs or the tiles, the Transmuter is configured to private-private mode.

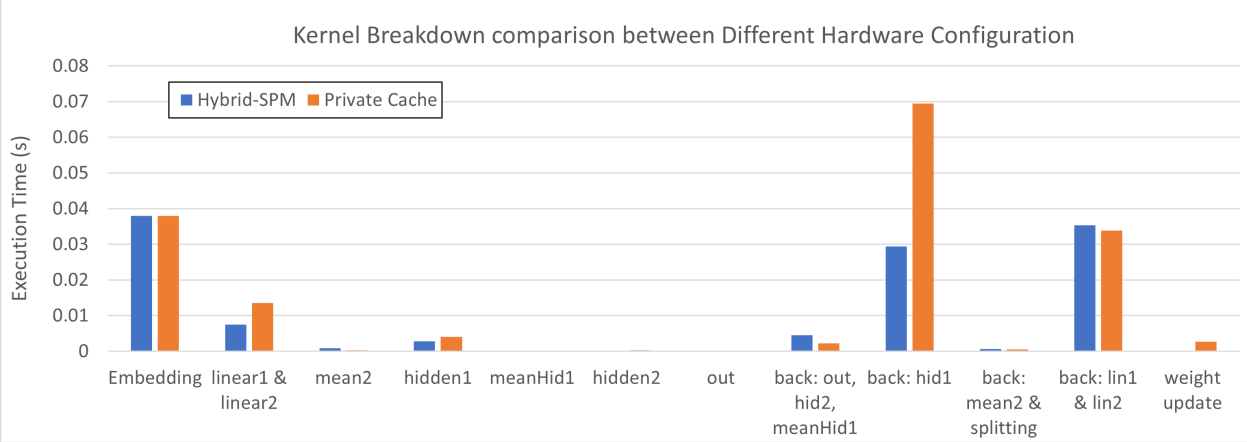


Figure 5.7: Performance comparison between hybrid mode and private cache mode for different kernels of Graphsage.

5.2.6 Dynamic Reconfiguration

The Graphsage workload has two hardware configurations that are optimal at different points of the kernel: private cache mode and hybrid scratchpad mode. The hybrid scratchpad mode configures half of the L1 cache into a shared scratchpad and the remaining half into a shared cache, while the L2 cache is configured to a private cache. The hybrid mode is the most suitable for the dense matrix-matrix multiplication kernels that are present in the linear layers in the forward pass, as well as the backpropagation calculation involving the weight matrix. The private cache mode where both the L1 and the L2 are configured to private cache is the most optimal configuration for

the rest of the Graphsage workload that do not utilize the scratchpad. Dynamic reconfiguration by switching to the most optimal hardware configuration for each of the kernels.

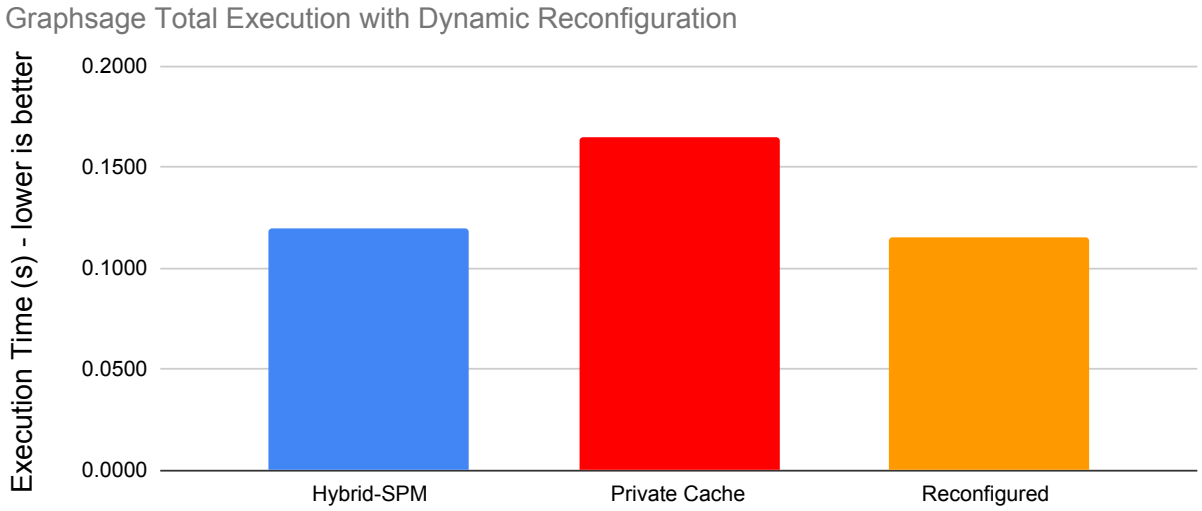


Figure 5.8: Overall performance of Graphsage on Transmuter with dynamic reconfiguration. The hardware switches between private cache mode and hybrid mode for different parts of Graphsage.

Figure 5.7 shows the performance comparison between the private cache mode and the hybrid mode for the different kernels of the Graphsage workload. The hybrid mode performs the best for the linear and hidden layers of the forward pass and backpropagation kernels, as well as the weight update. The private cache mode is preferred for the rest of the computation. Figure 5.8 shows dynamic reconfiguration results in speedup of 4% over the hybrid mode.

5.2.7 Optimizing for Energy Efficiency

Frequency and voltage scaling was used to maximize the energy efficiency of the workload. To find the optimal frequency point that consume the least amount of power, the Transmuter system was swept from 100MHz to 1.0GHz as shown in Figure 5.9 . For the Graphsage neural network computation, Transmuter is most energy efficient at 100MHz for 12.84 GFLOPS/W, and the efficiency decrease at higher frequency due to the high power consumption of the platform.

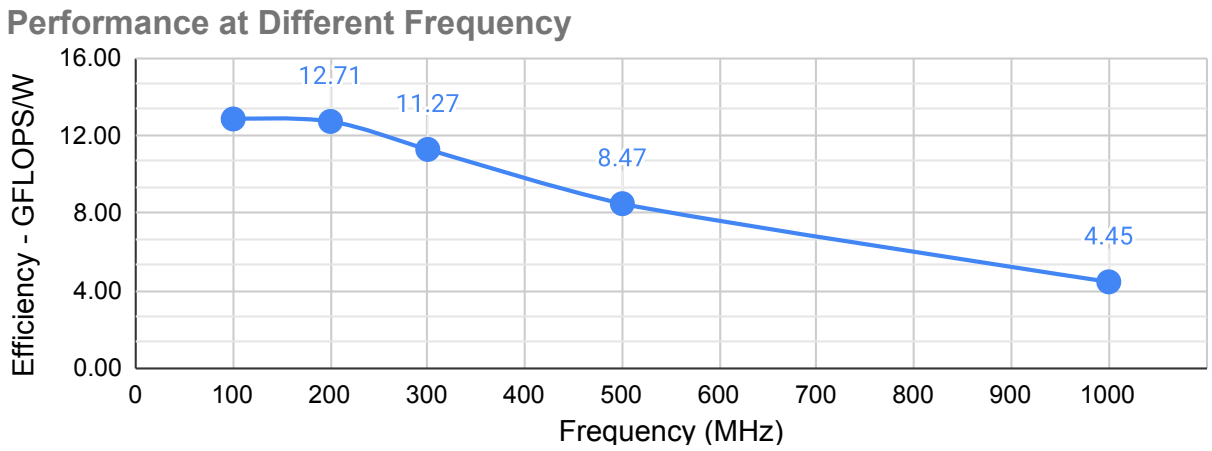


Figure 5.9: Frequency sweep of Graphsage workload. Most optimal performance of 12.84 GFLOPS/W is achieved at 100MHz.

CHAPTER 6

Sinkhorn Distance

6.1 Motivation

Fast and efficient information processing is critical to processing the vast amount of data that is available in today's computing landscape. Word Mover's Distance (WMD) algorithms, such as the Sinkhorn Distance, quantifies the similarities of two different text documents by calculating their relative distance from each other. In recent years, these algorithms have been especially useful in Natural Language Processing [76, 15]. Furthermore, Sinkhorn Distance has been a fitting case study to showcase the reconfigurability of the Transmuter [53].

6.2 Algorithm Overview

Sinkhorn distance is an optimal transport algorithm that helps determine the similarity of various documents [38] [12]. The query document and database are vectorized using pre-trained word embedding and a bag of words (BoW). A text document is represented by a BoW vector **d**, which is composed of $d_i = \frac{c_i}{\sum_{j=1}^{n} c_j}$ where the word *i* appears c_i times in the document. Each BoW vector is of size n, and is a sparse vector because a text document only features a limited number of words from the embedding matrix. The distance matrix *M* is a dense, $n \times n$ square matrix that represents the distance between each word, and is derived from the pre-trained word embeddings.

The high-level overview of the Sinkhorn algorithm is shown in Algorithm 2. The inputs of the algorithm is the query, which is a $n \times 1$ BoW vector, and data, a $n \times k$ matrix that contains

Algorithm 2 Sinkhorn Distance (MATLAB syntax)

function SINKHORN(query, data, M, γ, ϵ)
  $I = (query > 0); query = query(I); \tilde{M} = M(I, :);$ ▷ M: distance matrix
  $o = \texttt{size}(\tilde{M}, 2);$
  $H = \texttt{ones}(\texttt{length}(query), o)/\texttt{length}(query);$
  $K = \texttt{exp}(-\tilde{M}/\gamma);$ ▷ γ: regularization parameter
  $\tilde{K} = \texttt{diag}(1./query)K;$
  $U = 1./H;$
  $err = \infty;$
  while $err > \epsilon$ **do** ▷ ϵ: tolerance
    $V = data./(K'U);$ ▷ Masked-GeMM
    $U = 1./(\tilde{K}V);$ ▷ DMSpM
    $err = \texttt{sum}((U - U_{prev})^2)/\texttt{sum}((U)^2);$
  end while
  $D = U. * ((K. * \tilde{M})V);$
return $\texttt{sum}(D)$ ▷ Sinkhorn distances between `query` and `data`
end function

k documents each represented by its own BoW vector. The distance matrix M, regularization parameter γ, and convergence tolerance ϵ are all constants. The typical dimensions and density of each matrix in the algorithm after the first 100 iterations are shown in Table 6.1. The vectors that represent the BoW vectors of text documents (query and data) are quite sparse at approximately 1%. However, the high density of matrix H and other variables that are derived from it prevents the computation from becoming completely sparse. The inner-loop of the algorithm is comprised of two main kernels: a dense matrix-matrix operation ($K'U$) masked by a sparse matrix ($data$), and a dense matrix - sparse matrix multiplication ($\tilde{K}V$).

Matrix	Dimensions	Density
query	(8000,1)	1%
data	(8000,1000)	1%
M	(8000,8000)	99%
$\tilde{M}$,K	(65,8000)	13%
H, U	(65,1000)	84%
V	(8000,1000)	1%

Table 6.1: Dimensions and densities of the input matrices and internal variables in the Sinkhorn workload after 100 iterations.

6.3 Masked Dense Matrix-Matrix Multiplication (Masked-GeMM)

In a typical dense matrix-matrix multiplication, every element of each matrix needs to be accessed multiple times to produce the result matrix. This is an extremely intensive procedure, where the cost of computation scales by $O(n^3)$ for matrices of the size *n*x*n*. The dense matrix-matrix multiplication kernel comes from the equation:

$$V = data./(K'U) \tag{6.1}$$

Matrix *K* is a rectangular matrix calculated during the initial pre-processing, with a density of roughly 13%. Matrix *U* is a dense, rectangular matrix that is updated at each iteration. The change in density of Matrix *U* after each iteration is shown in Figure 6.1.

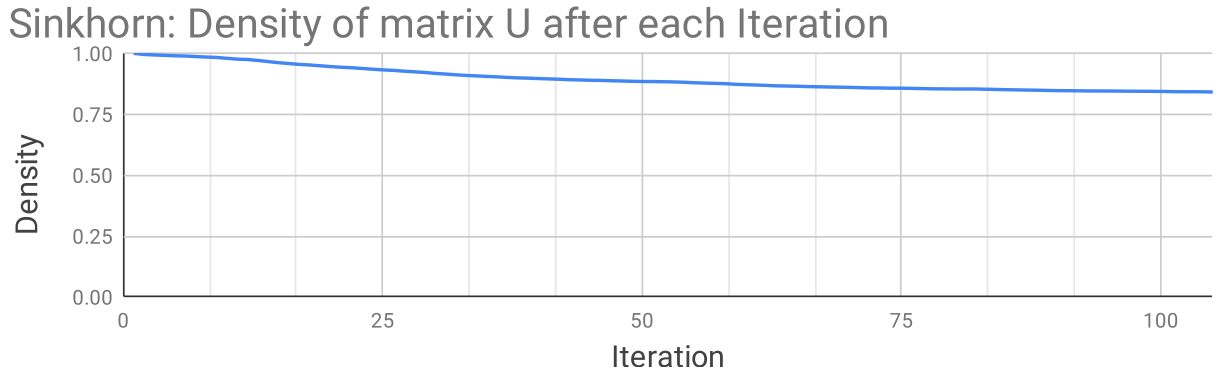


Figure 6.1: Progression of the density of matrix U after each iteration of Sinkhorn algorithm. Matrix U starts from density of 1.0 and reaches 0.84 after 100 iterations.

Matrix *U* starts as a fully dense matrix, but becomes progressively sparser after each iteration, and reached a density of 0.84 at one hundred iterations. While there is some variation in density of matrix *U*, neither matrix *K* nor matrix *U* are sparse enough to apply a sparse matrix multiplication algorithm. However, because the result of this dense matrix-matrix multiplication (GeMM) undergoes an element-wise division with the sparse matrix, *data*. As such, only the matrix entries that corresponds to the same position as the nonzero elements of *data* are relevant for the final

matrix *V*. Thus, matrix *data* can be used as a mask to filter the majority of computation that are present in the GeMM of $K'U$ and significantly reduce the computation. Because of the masking, the resulting matrix *V* is a sparse matrix with similar sparsity as the *data* matrix.

6.4 Dense Matrix - Sparse Matrix Multiplication (DMSpM)

The dense matrix - sparse matrix multiplication kernel of the Sinkhorn workload comes from the equation:

$$U = 1./(\tilde{K}V) \tag{6.2}$$

Matrix $\tilde{K}$ is a constant rectangular matrix calculated during the initialization phase of the workload, and depends on the *query* input. Matrix *V* is a sparse matrix that is the product of Equation 6.1. As such, the computation is a dense matrix multiplied by a sparse matrix. Because a conventional dense matrix multiplication algorithm would be too costly, a sparse matrix multiplication algorithm was tweaked to compensate for the dense nature of the first operand.

Two different variations of DMSpM were tested: inner-product and outer-product. In the inner-product DMSpM algorithm, each GPE is allocated a row of the dense matrix and a column of the sparse matrix as shown in Fiture 6.2. Each GPE performs the vector multiplication. Because the row vector is dense while the column vector is sparse, only the non-zero elements of the column vectors are multiplied with the values in the corresponding positions of the dense row. Once a GPE completes its assigned workload, the result is sent to the LCP, which commits the final result to memory.

The outer-product DMSpM algorithm is a variation of the outer-product sparse matrix-matrix multiplication. The algorithm is split into two phases, DMSpM-Multiply and DMSpM-Merge, as shown in Fiture 6.3. In the DMSpM-Multiply, the columns of the dense matrix and the rows of the sparse matrix are multiplied to generate the partial-product matrices. Unlike a typical outer-

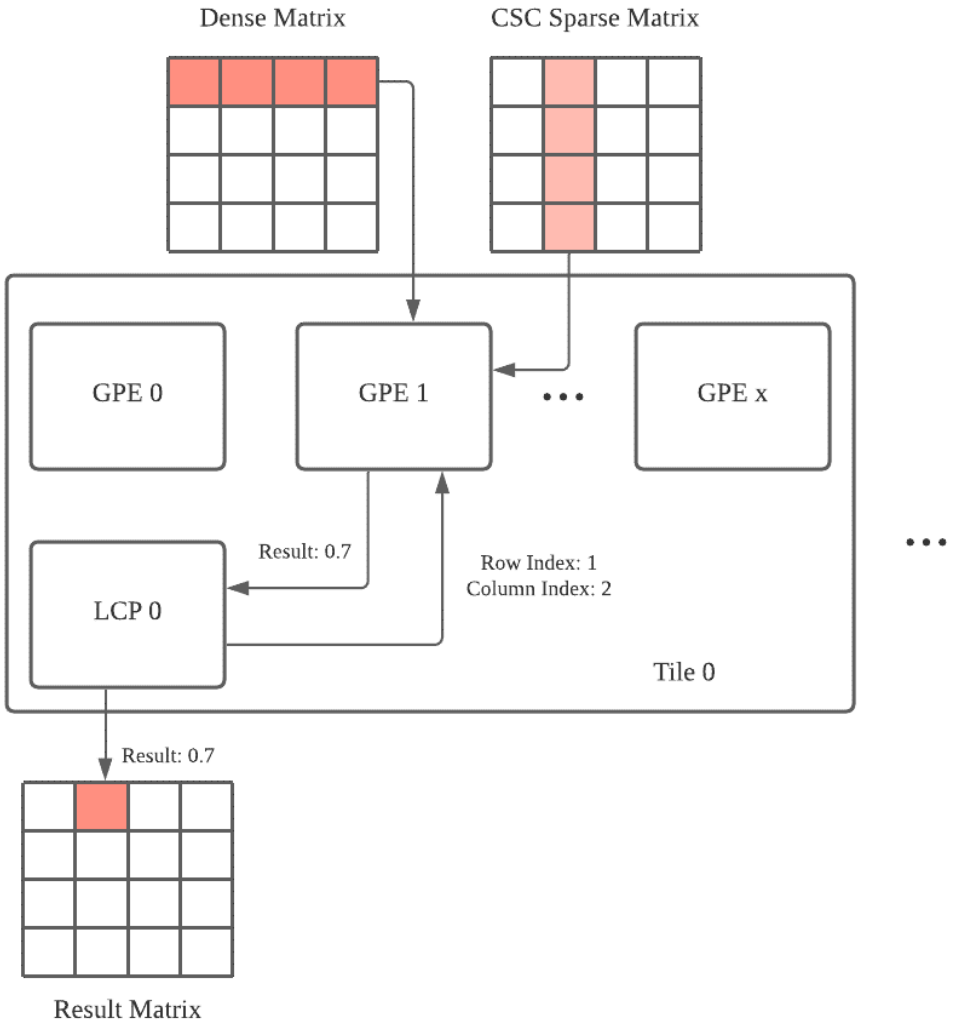


Figure 6.2: Diagram of inner-product based DMSpM.

product algorithm, where a sparse column is used to allocate work across the GPEs, instead of allocating work based on the column vectors, the work is assigned to the GPEs based on the sparse row vectors. Once all the partial-products are generated, DMSpM-Merge is conducted in the same manner a a regular outer-product merge.

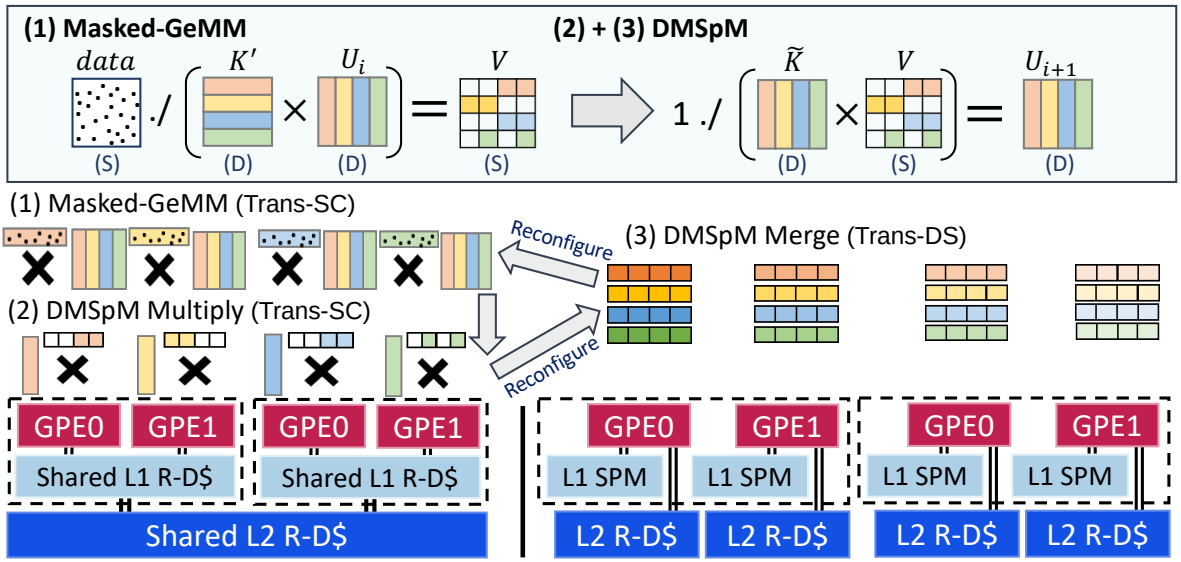


Figure 6.3: Sinkhorn mapping on Transmuter. Sparse (S)/Dense (D) indicate the nature of inputs. Sinkhorn iterates between Masked-GeMM and DMSpM, with reconfiguration before and after DMSpM-Merge.

6.4.1 Dynamic Reconfiguration

The most optimal Transmuter configuration for Masked-GeMM and DMSpM-Multiply stages of the Sinkhorn algorithm is the shared cache configuration. On the other hand, the DMSpM-Merge performs the best with the use of a private scratchpad memory to maintain the sorting list for each PE. The dynamic reconfiguration feature allows Transmuter to optimize its hardware for specific kernels. Since the algorithm needs to keep iterating until the error converges, there needs to be two reconfiguration process per iteration.

The scalability and benefits of implementing dynamic reconfiguration through the Transmuter architecture was explored by mapping Sinkhorn onto 2×16 and 4×16 systems. As shown in Figure 6.4, Masked-GeMM and DMSpM-Multiply exhibit the best performance in Trans-SC configuration, thanks to data reuse across GPEs during the multiplication. On the other hand, DMSpM-Merge has the optimal performance on Trans-DS configuration, exhibiting a 2.0× improvement over Trans-SC on both 2×16 and 4×16 systems. Thus, each iteration of Sinkhorn workload was computed by performing two hardware reconfigurations on-the-fly: Trans-SC →Trans-DS before the start of DMSpM-Merge, and Trans-DS →Trans-SC at the end of it, for the next Masked-

GeMM iteration. The reconfiguration time is $<$10 cycles and hence did not have perceptibly impact on the performance or energy of the overall execution.

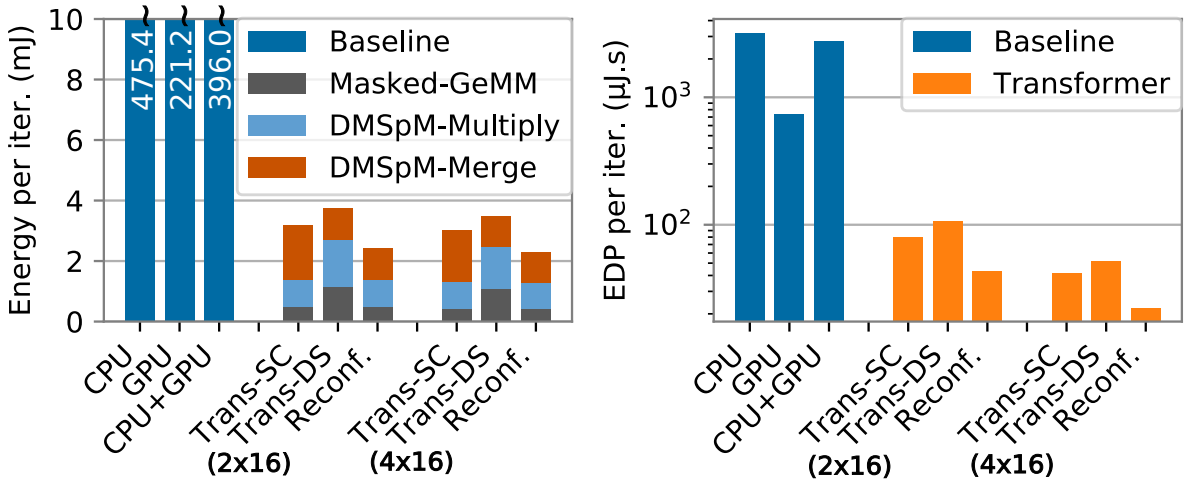


Figure 6.4: Energy and EDP comparisons of 2×16 and 4×16 Transmuter executing the three phases in Sinkhorn against the CPU, GPU and heterogeneous impl. *query: (8k×1), 1%, data: (8k×1k), 1%, M: (8k×8k), 99%*. Per inner-loop iteration energy (left) and EDP (right) comparing Trans-SC, Trans-DS and Reconf. (Trans-SC +Trans-DS).

Due to the lack of hardware consistency models in the Transmuter system, the L1 R-DCaches need to be flushed when transitioning from Trans-DS configuration to Trans-SC configuration (Trans-DS →Trans-SC). When configuration from Trans-SC mode to Trans-DS mode (Trans-SC →Trans-DS), both L1 and L2 caches need to be flushed before the reconfiguration API call to ensure all the updated values of matrix U have been committed to memory to be used in the next iteration. The total cache flush time accounts for only 0.2% of the total execution time.

Masked-GeMM, DMSpM-Multiply and DMSpM-Merge contribute to 18.5%, 36.8% and 44.7%, respectively, of the remainder. Overall, dynamic reconfiguration in 4×16 Transmuter results in 44.3% and 89.2% better performance and Energy-Delay Product (EDP), respectively, over Trans-SC-only configuration.

6.4.2 Comparison against CPU and GPU

On the CPU, Masked-GeMM and DMSpM takes 27.8% and 71.7% of the execution time, respectively. For the GPU, the GeMM and the sparse element-wise division are not fused. GeMM, sparse

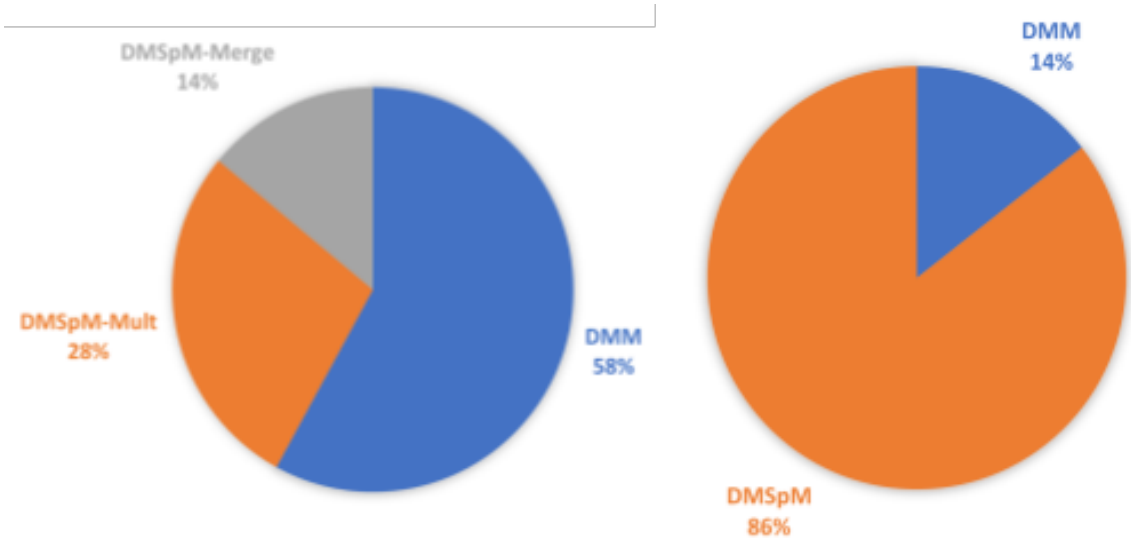


Figure 6.5: Kernel breakdown of Sinkhorn algorithm with outer-product and inner-product based DMSpM. Inner-product algorithm eliminates the DMSpM-Merge phase

element-wise division, and DMSpM on the GPU comprise 82.5%, 4.9% and 6.9%, respectively. A heterogeneous solution is also compared against, where the Masked-GeMM is done on the CPU and DMSpM on the GPU, but this implementation is bottlenecked by CPU→GPU data transfers. Figures 6.4 compares the energy and EDP of Transmuter against the CPU/GPU. The 4×16 Transmuter achieves 144.1× (206.8×) and 33.0× (96.2×) better EDP (energy) over the CPU and GPU, respectively. Transmuter is 172.34× more efficient than the heterogeneous solution.

CHAPTER 7

Genomic Sequencing

7.1 Motivation

Whole genome sequencing is the process of determining the complete DNA sequence, and it is a major part of computational biology. The gene sequencing pipeline is often broken into four primary steps: read, align, assemble, and analysis [10]. In the read phase, short fragments of gene sequences are sampled from the target organism through a sensor. Signal or image processing techniques are used on sensor data to obtain the protein codes of the sampled fragments.

In the alignment phase, the short sampled fragments are compared the reference genes in the known database using sequence alignment algorithms. Sequence alignment algorithms detect similar regions between two sequences and produce a similarity score for assessing how close they are. Smith-Waterman algorithm [78] is the most well known alignment algorithms, and it used as the basis for most modern alignment tools, such as BLAST [33], Bowtie2 [39], or BWA-SW [40].

The assemble phase uses the sampled fragments to construct the full sequence based on the alignment data. Finally, the analysis phase searches the assembled sequence for variants or abnormalities that are used to diagnose the target organism. Statistical techniques or machine learning algorithms such as deep neural networks and hidden Markov models are commonly used for these analyses.

Sequence alignment process takes up a significant portion of a whole genome sequencing pipeline. The open-source genome analysis platform, SpeedSeq [10] takes 13 hours to fully align

and annotate a human genomes with 50x coverage, 61.5% (8 hours) of which are spent during the alignment process. There have been several prior works to speedup the Smith-Waterman alignment algorithms in various hardware platforms. SWPS3 [68] accelerates Smith-Waterman through multi-threading and SIMD vector instructions in Intel's x86 or IBM's Cell architecture. CUDASW++ [44] is a CUDA-implementation of the algorithm, and Altera provides an FPGA-implementation for their XD1000 platform [77].

Several parts of the text and graphs for this chapter were taken from the prior publication on IEEE Conference on Cluster Computing (CLUSTER) in 2017 [55].

7.2 Smith-Waterman Alignment Algorithm

Smith-Waterman is a dynamic programming algorithm that produces local, pairwise alignment between two string sequences. Smith-Waterman takes two string sequences that are in FASTA format: a reference sequence and a query sequence. The reference sequence is the baseline sequence that comes from the gene database. The query sequence is the sampled fragment that needs to be aligned to the reference. Typical length of the query sequence range from 30 base-pairs (bps) to 400 bps, depending on the technology used for sampling the sequence. Smith-Waterman outputs the segment of the two sequences that has the best match and the alignment score that indicates how similar the two segments are. The alignment process has two main parts to the alignment process: generating the scoring matrix, and back-traversing the largest score.

7.2.1 Scoring Stage

The scoring matrix is a dynamically generated m+1 by n+1 matrix, where m is the length of the reference sequence and n is the length of the query sequence, as shown in Figure 7.1a. The first row and column of the matrix are initialized to zero, and the matrix is filled by traversing through the matrix from the top-left corner.

Given two sequences a and b, the score of each matrix, $H(m,n)$ is calculated as shown:

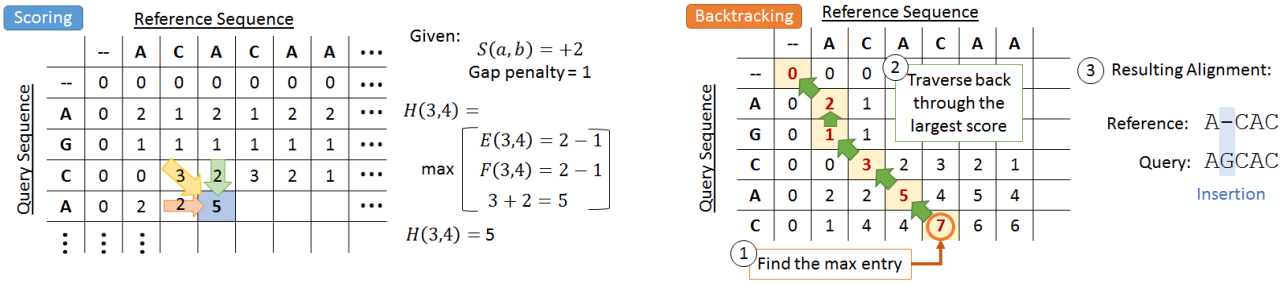
$$H(m,n) = max \begin{cases} E(m,n) \\ F(m,n) \\ H(m-1,n-1) + S(a_m, b_n) \end{cases}$$

$$E(m,n) = max \begin{cases} H(m,n-1) - g_o \\ E(m,n-1) - g_e \end{cases}$$

$$F(m,n) = max \begin{cases} H(m-1,n) - g_o \\ F(m-1,n) - g_e \end{cases}$$

The constants, g_o and g_e, are gap penalties that compensate for insertions or deletions in the sequence. $S(a_m, b_n)$ is the substitution matrix that describes the rate in which amino acid denoted by a_m transitions to b_n.

To calculate the score at position (m,n), values from three adjacent position needs to be compared: horizontal $(m-1,n)$, vertical $(m,n-1)$, and diagonal $(m-1,n-1)$. The $E(m,n)$ and $F(m,n)$ values derived from the horizontal and vertical positions represent a misalignment that skips a base in the query or the reference sequence. In both instances, the values coming from adjacent entries are subtracted by the gap penalty to take compensate for the mismatch. The direct



(a) Scoring Stage

(b) Backtracking Stage

Figure 7.1: Smith-Waterman Alignment Overview

one-to-one alignment is represented by the diagonal value, $H(m-1, n-1) + S(a_m, b_n)$.

7.2.2 Backtracking Stage

Once the full matrix is created, the algorithm finds the entry with the largest score, as shown in Figure 7.1b. Starting from this maximum entry, the algorithm looks at top top, left or top-left adjacent entries and tracks back to the one with the largest score. A diagonal movement indicates a match, horizontal movement implies a deletion, and vertical movement denotes a insertion on the query sequence. The process is repeated until a cell with zero value is reached, and the resulting continuous path describes the location and characteristic of the most optimal alignment discovered by the algorithm.

7.3 Scalable Vector Architecture

ARM's Scalable Vector Extension (SVE) was used to explore the impact of varying the SIMD vector length in Smith-Waterman workload. Unlike existing SIMD architectures such as ARM's Advanced SIMD (more commonly known as NEON [1]), SVE proposes a flexible SIMD ISA that is vector-length agnostic: the same binary code can be run across hardware implementations with different data widths. With SVE, the programmer can scale their SIMD code to larger vector lengths without the need of rewriting the code. SVE architecture comes with its own 32 vector registers, as well as 16 predicate registers that can configure and mask the SIMD pipeline. The logical and arithmetic SVE operations are executed in the SVE hardware pipeline. In the baseline implementation of SVE, the memory instructions are broken down into regular ARM micro-ops and processed by the host processor's memory unit, but more advanced implementations can improve vector memory operations through memory coalescing.

7.4 Mapping Smith-Waterman Algorithm to Scalable Vector Engine

The majority of the computation time in Smith-Waterman algorithm is spent generating the scoring matrix. For a reference sequence of length m and query sequence of n, the matrix computation time scales by $O(mn)$. We accelerated the scoring matrix computation in vector architecture using three different methods: batch, sliced, and Wavefront.

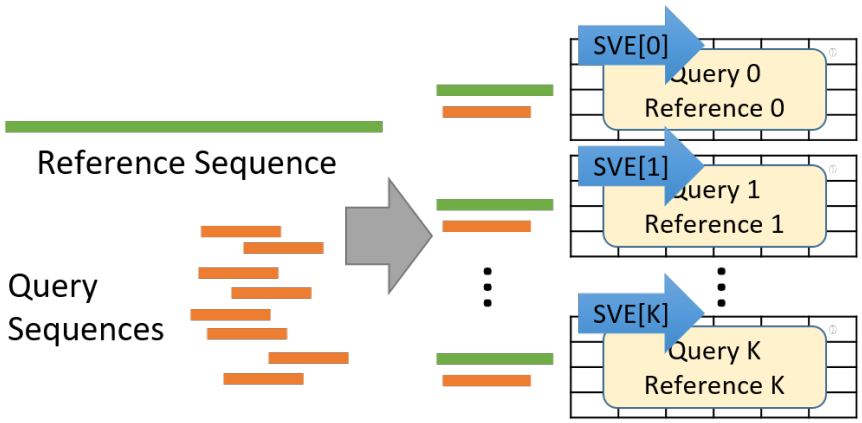
7.4.1 Batch Smith-Waterman

The batch implementation accelerates the sequence alignment process by operating on multiple reference-query pairs at once. Given a pool of query sequences that need to be aligned, the algorithm distributes a query to each vector as shown in Figure **??**, and forms a batch of queries to be processed together. The number of reference-query pairs that the SIMD unit can process depends on the width of the vector unit. All the vectors operate synchronously, but do not have any data dependencies between each other. This optimization focuses on improving the throughput, while keeping the processing latency bound by the largest length query sequence in the batch.

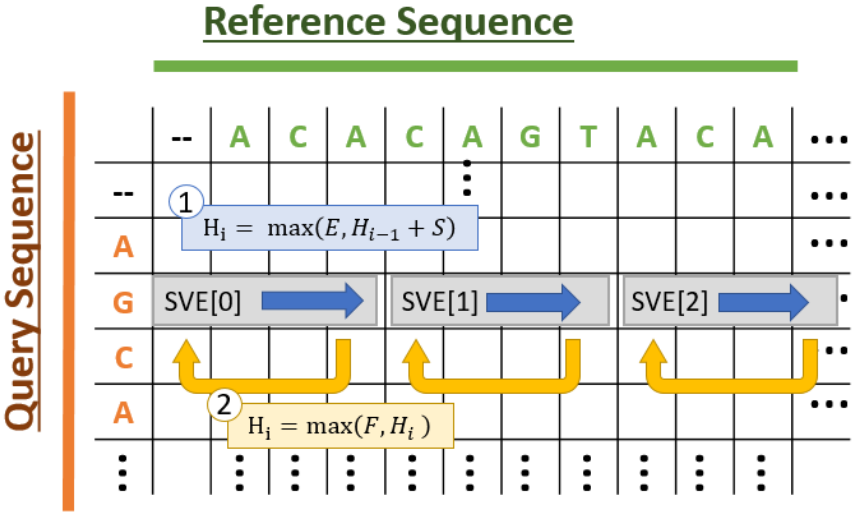
7.4.2 Sliced Smith-Waterman

The sliced algorithm (or more commonly referred to as "Striped" [18]) accelerates the Smith-Waterman algorithm itself by vectorizing parts of score matrix computation. The reference sequence is partitioned by the number of vectors available, and each vector operates only on its partitioned segment, as shown in Figure **??**.

For each row m of the matrix computation, the scores with vertical ($E(m, n)$) and diagonal ($H(m-1, n-1)$) data dependencies are calculated first. Then the algorithm makes a second pass through the row to resolve the horizontal data dependency, $F(m, n)$. During this second pass, the algorithm checks whether the horizontal component $F(m, n)$ is bigger than the similarity score of current entry.



(a) Batch Smith-Waterman Overview



(b) Sliced Smith-Waterman Overview

Figure 7.2: Batch and Sliced Vectorization of Smith-Waterman

If the horizontal component is smaller, then all the subsequent scores do not need to be updated because the horizontal dependency has no impact on the future entries. Therefore, the horizontal update can be bypassed, reducing the overall computation by a significant amount for long sequences that resolve the dependency early. The slicing help maximize the data reuse for a given row of score matrix at the cost of increase in worst-case computation introduced by the additional pass through the row [78].

7.4.3 Wavefront Smith-Waterman

The Wavefront implementation traverses the similarity matrix diagonally, so that each iteration depends solely on the results from the previous iterations [67]. The algorithm starts from the top-left entry in the top-left corner of the similarity matrix, and travels across the matrix with each iteration until the whole matrix is filled. Given the entry m in the n-th diagonal, the horizontal $F(m, n)$ and vertical $E(m, n)$ dependencies come from the entries in the $n-1$-th diagonal, and the diagonal dependency $H(m-1, n-1)$ comes from the $n-2$-th diagonal as shown in Figure 7.3.

At each iteration, the diagonal strip of entries keeps growing by one until the length is equal to the length of the smaller of the two input sequences. Each diagonal strip is partitioned by the vector width. The vectors will be underutilized during the early stages of the execution due to the number of operating entries being less than the available vector length, but lanes will remain at high utilization for majority of execution time for long input sequences.

The Wavefront algorithm resolves the data dependencies of the Smith-Waterman algorithm by scheduling the execution along the diagonal so only the entries that have the all the available dependencies are executed. In return, the addressing of the memory operations are more fine grained than the continuous or strided accesses of bulk and sliced implementations.

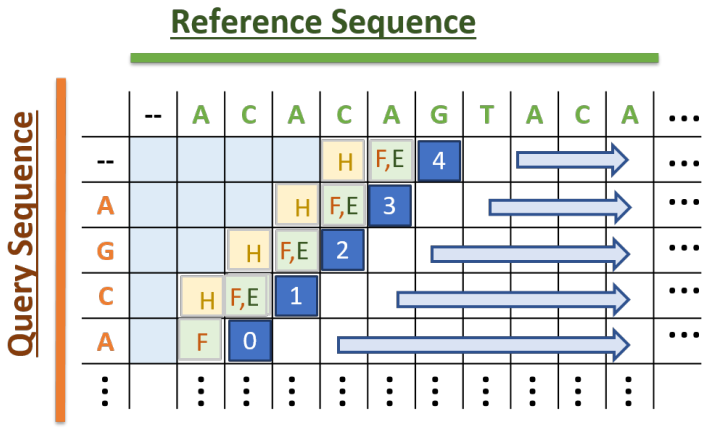


Figure 7.3: Wavefront Smith-Waterman Overview

7.5 Experimental Results

The performance of Smith-Waterman algorithm at varying vector length was compared for each of the three different vectorization schemes. The study was conducted using a custom version of gem5 with SVE simulation provided by ARM. The baseline architecture is an out-of-order 64-bit ARM core with 8-wide issue and 1GHz clock. The fixed-vector SIMD architecture (NEON) has data width of 128-bits, while the SVE architecture has four different data widths ranging from 128-bits to 1024-bits. Due to the lack of an available C compiler for SVE, the SVE implementation was hand-written in assembly code. The detailed simulation specification is given in Table 7.1.

Table 7.1: Gem5 Simulation Specification for Vector Reconfiguration

Component	**Configuration**
Core	Single-Core out-of-order 64-bit ARM, 1GHz, 8-issue SIMD Width: 128-bit (NEON), 128-1024-bit (SVE)
Cache	32KB private L1 instruction cache, 2-way associative 64KB private L1 data cache, 2-way associative 4MB private L2 inclusive cache, 8-way associative
DRAM	Capacity: 8GB Latency: 30 ns Memory Controller Bandwidth: 12.8 GB/s
Benchmark	Batch and Sliced Smith-Waterman Reference: 25-400bp sections from *E.Coli 536* Strain Query: 1000x 25-400bp

7.5.1 Comparison of Vector Mapping

The performance of Batch, Sliced, and Wavefront Smith-Waterman are compared in Figure **??** and Figure **??**. The sequencing times are taken from SVE 512-bit configuration with 64kB of L1 Data cache. The Batch Smith-Waterman outperforms both Sliced and Wavefront solutions for queries smaller than 50 base pairs. Wavefront algorithm exhibits better speedup between sequences of length 50 and 200, while Sliced Smith-Waterman starts to outperform the rest heavily beyond 400 base pairs. The simplicity and low overhead of the Batch implementation help the algorithm perform favorably for short sequences, while Wavefront implementation performs favorably at

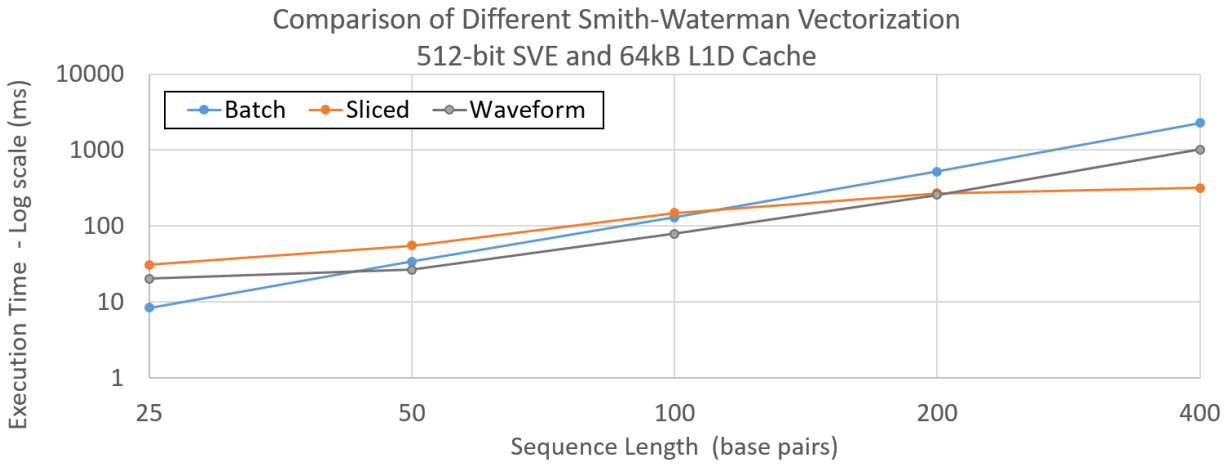


Figure 7.4: Runtime of Different Smith-Waterman algorithm over Batch implementation for SVE 512-bit with 64KB L1 Data Cache System. Sequencing time is the total time required for aligning 1000 queries.

medium-length sequences due to its efficient utilization of the SIMD hardware.

The Sliced Smith-Waterman performs poorly at low sequence lengths due to the high overhead of execution, but scales favorably at larger sequence lengths due to the bypassing of the horizontal dependency calculation. At low sequence lengths, Sliced Smith-Waterman only has a limited window where bypassing helps reduce the computation overhead of the algorithm. At short sequence lengths, Sliced Smith-Waterman can perform worse than even a naïve CPU implementation, due to the performance overhead of resolving horizontal dependency by traversing the same row twice. But at longer sequences, it is more likely for the bypassing to be triggered early on in the re-computation step, resulting in significant amount of computation savings. This is also the reason Sliced algorithm struggles to take advantage of the wider vectors. Wider vectors help the Sliced algorithm compute the initial calculations for diagonal and vertical dependencies faster, but the increased segmentation reduces the bypassing window, resulting in a performance decrease for short and medium sized sequences (less than 200 bps).

The memory bandwidth utilization shows the strength of Sliced and Wavefront implementation in minimizing the memory overhead. Figure 7.6 show the average read and write memory bandwidth of the three implementations at different sequence lengths. The memory bandwidth demands

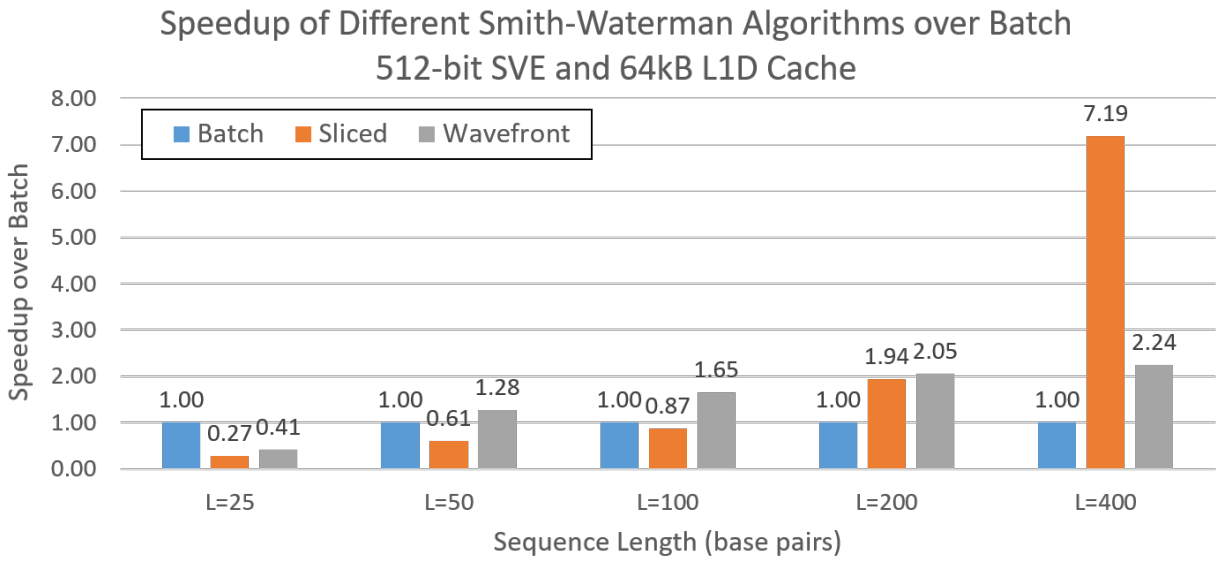
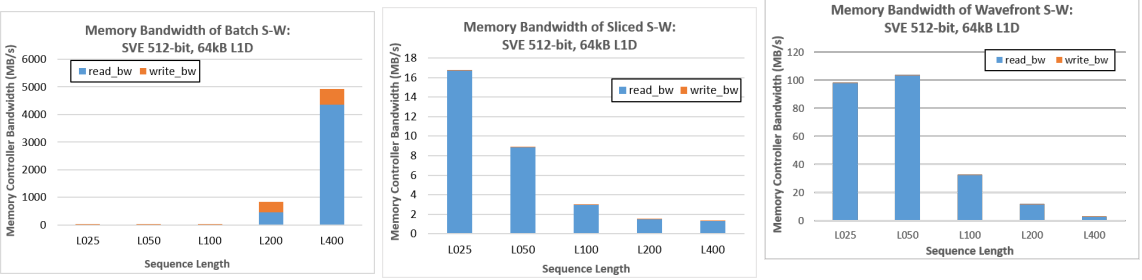


Figure 7.5: Speedup of Different Smith-Waterman algorithm over Batch implementation for SVE 512-bit with 64KB L1 Data Cache System. Batch is favored for sequences below 50 bps, Wavefront is favored between 50-200 bps, and Sliced is favored for sequences 400 and longer.

of Batch algorithm increase dramatically with larger sequence sizes and start to become a performance bottleneck. On the other hand, Sliced and Wavefront implementations manage to reduce the memory bandwidth with increasing sequence length. This is because both Sliced and Wavefront Smith-Waterman implementations only operate on a single query at a time and the dependent data required to perform matrix calculation are only accessed once. Improvements in data reuse help minimize bandwidth overhead in Sliced and Wavefront optimization, and lead to improved performance. Comparing Sliced and Wavefront algorithms, Sliced algorithm require significantly less memory bandwidth compared to the Wavefront algorithm. This is because Sliced algorithm accesses the sequences more efficiently by operating on contiguous blocks of memory during its execution, while Wavefront algorithm makes heavy use of fine-grained data accesses, causing the memory controller to request multiple cache lines at ones where only a few portion of the data was actually requested.



(a) Batch Memory Bandwidth (b) Sliced Memory Bandwidth (c) Wavefront Memory Bandwidth

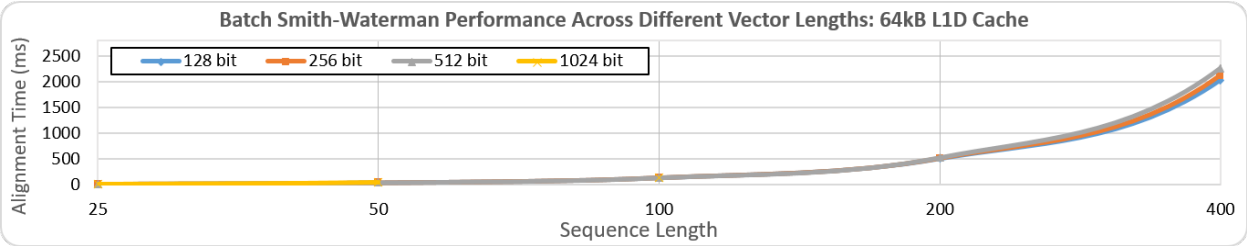
Figure 7.6: Average Read and Write Memory Bandwidth of Smith-Waterman for Batch, Sliced, and Wavefront Optimization. Maximum bandwidth of the System is $12.8GB/s$. Sliced and Wavefront makes more efficient use of memory bandwidth because they operate only on a single sequence pair at a time.

7.5.2 Vector Width Reconfiguration

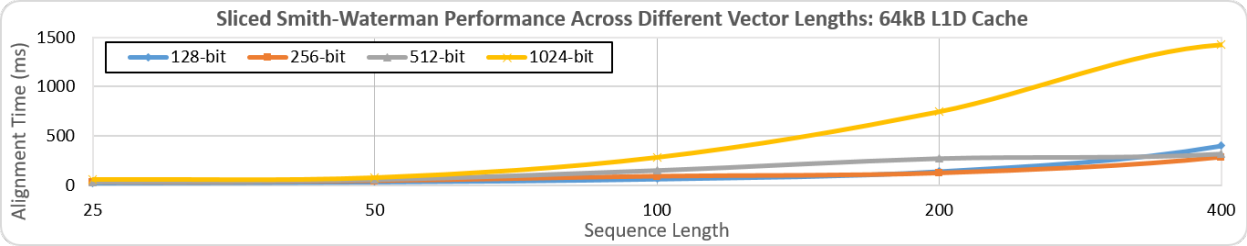
The sequencing time of Batch, Sliced, and Wavefront implementations at different SVE vector widths ranging from 128-bit to 1024-bit are shown in Figure 7.7. Batch Smith-Waterman shows an exponential increase in alignment time with increasing sequence length for all vector widths, and the execution time remains relatively the same regardless of vector width, with larger vectors performing worse under most circumstances. This is due to algorithm's high dependence on vector memory operations, which are not improved with vector width as SVE simply breaks down vector loads and stores into individual micro ops. On the other hand, alignment time of Sliced algorithm does not grow as rapidly as the Batch solution. While large SVE vectors performs poorly for small length queries, the wide 512-bit vector do not degrade in performance as rapidly as the smallest 128-bit configuration.

Unlike Batch and Sliced implementations, the Wavefront algorithm exhibits strong performance scaling with increase in vector length, especially for long sequences. Doubling the vector width from 128-bit to 256-bit leads to 1.3-1.8x speedup depending on the sequence length. While the performance gain shows a diminishing return with each additional SIMD lane, the Wavefront algorithm takes full advantage of the additional hardware, thanks to the algorithm's handling of data

(a) Batch Performance



(b) Sliced Performance



(c) Wavefront Performance

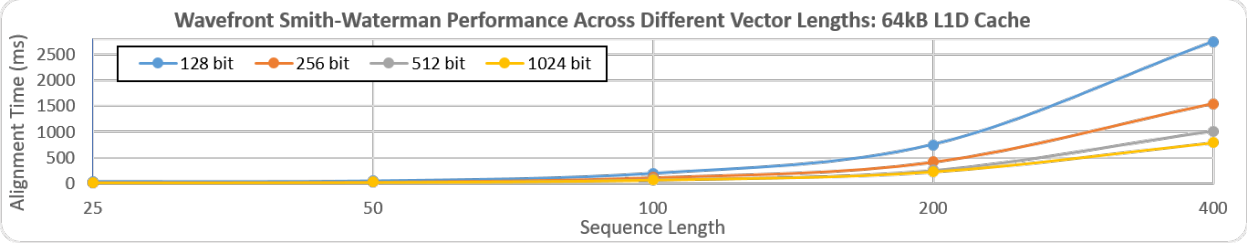


Figure 7.7: Performance of Smith-Waterman for Batch, Sliced and Wavefront Optimization.

dependency in its execution.

7.6 Evaluation

Based on the experimental data, one can map the optimal algorithm and SVE vector width combination at each sequence length as shown in Table 7.2. For large sequence lengths, our solution is able to achieve up to 53.9x speedup over the CPU solution. Dynamically selecting the algorithm and reconfiguring the SIMD width depending on the input sequence length leads to up to 4x

performance speedup over a homogeneous solution.

Table 7.2: Optimal Algorithm and Vector Width Selection at Each Sequence Length

Sequence Length	**Algorithm**	**SVE Width**	**Speedup over CPU**	**Speedup over Sliced 256-bit**
25 bps	Batch	128-bit	10.8	3.7
50 bps	Wavefront	1024-bit	12.0	1.7
100 bps	Sliced	128-bit	16.7	1.4
200 bps	Sliced	256-bit	32.5	1.0
400 bps	Sliced	256-bit	53.9	1.0

CHAPTER 8

Conclusion

The ubiquity of the internet and the explosive growth of compute power fueled by the Moore's law led to enormous shifts in the size of data that gets computed on a daily basis. While the improvements in compute power and efficiency has stagnated over the years, workload sizes have continued to get larger and larger. Today's datasets are not only big, they are also very sparse, with majority of its content being comprised of zeros [23, 48]. To achieve high performance and efficiency on these large, sparse datasets, it is critical to tune existing algorithms to take advantage of their sparsity. This is especially important for linear algebra kernels that can exhibit performance differences between sparse and non-sparse versions in the order of several magnitudes [52].

In this thesis, we explored two different approaches to sparse matrix-matrix multiplication: outer-product and row-wise. Outer-product approach seeks to minimize the memory accesses to the input matrices at the cost of having to manage large partial-product matrices. We were able to demonstrate outer-product approach achieving high performance and efficiency on OuterSpace accelerator with coalescing crossbar and a programmable prefetcher. However, high overhead of the partial-products incur a high penalty especially on platforms with limited local memory. Row-wise approach mitigates this overhead by limiting the partial-products that is generated to a single row. We were able to show row-wise approach achieving better performance than outer-product approach on Transmuter, a reconfigurable architecture with register-to-register communication capability. The row-wise approach has three different implementations for the merging of the partial-products: dense vector, sorting list, and systolic. Sorting list tend to perform better

at lower densities, while dense vector tend to perform better on denser matrices. For rectangular matrices with small widths, systolic merge can be the best option at moderate densities, and the performance can be further improved by reordering the matrices to have more clusters. We show that row-wise algorithm can achieve the best performance when it is tuned based on the input matrix dimension and density.

In addition to sparse linear algebra, three other kernels were explored to showcase the benefits of reconfigurable hardware: Graphsage deep neural network, Sinkhorn algorithm for optimal transport distance, and Fast-Fourier Transform (FFT). Graphsage and Sinkhorn Distance both take advantage of Transmuter's reconfigurable memory by executing parts of the kernel with high memory reuse in scratchpad mode and the rest in cache mode. Fast-fourier transform makes use of the register-to-register communication to organize the processing elements of the Transmuter in 1D systolic fashion, and assigning each stage of the FFT to a GPE. Overall, we were able to showcase how dynamic reconfiguration allows the programmer to improve their workload by utilizing a variety of different hardware modes, each optimized for specific parts of the kernel.

BIBLIOGRAPHY

[1] Introducing NEON™ Development. *ARM Limited*, 2009.

[2] M. Anders, H. Kaul, S. Mathew, V. Suresh, S. Satpathy, A. Agarwal, S. Hsu, and R. Krishnamurthy. 2.9tops/w reconfigurable dense/sparse matrix-multiply accelerator with unified int8/inti6/fp16 datapath in 14nm tri-gate cmos. In *2018 IEEE Symposium on VLSI Circuits*, pages 39–40, June 2018.

[3] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–31, 2016.

[4] E. O. Brigham and R. E. Morrow. The fast fourier transform. *IEEE Spectrum*, 4(12):63–70, 1967.

[5] Aydin Buluc and John R Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In *2008 37th International Conference on Parallel Processing*, pages 503–510. IEEE, 2008.

[6] Aydin Buluç and John R Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing*, 34(4):C170–C191, 2012.

[7] Aydin Buluç and John R Gilbert. On the representation and multiplication of hypersparse matrices. In *IEEE Int'l Symposium on Parallel and Distributed Processing*, IPDPS '08, pages 1–11. IEEE, 2008.

[8] Anurat Chapanond, Mukkai S. Krishnamoorthy, and Bülent Yener. Graph Theoretic and Spectral Analysis of Enron Email Data. *Computational & Mathematical Organization Theory*, 11(3):265–281, Oct 2005.

[9] Yifeng Chen, Xiang Cui, and Hong Mei. Large-scale fft on gpu clusters. ICS '10, page 315–324, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Colby Chiang, Ryan M Layer, Gregory G Faust, Michael R Lindberg, David B Rose, Erik P Garrison, Gabor T Marth, Aaron R Quinlan, and Ira M Hall. SpeedSeq: ultra-fast personal genome analysis and interpretation. *Nature Methods*, 12(November 2014):1–5, 2015.

[11] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, page 157–172, New York, NY, USA, 1969. Association for Computing Machinery.

[12] 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.

[13] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations, 2015. Version 0.5.1.

[14] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):25, 2015.

[15] Yihe Dong, Yu Gao, Richard Peng, Ilya Razenshteyn, and Saurabh Sawlani. A study of performance of optimal transport, 2020.

[16] R. Dorrance and D. Markovic. A 190gflops/w dsp for energy-efficient sparse-blas in embedded iot. In *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pages 1–2, June 2016.

[17] 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.

[18] Michael Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007.

[19] Siying Feng, Jiawen Sun, Subhankar Pal, Xin He, Kuba Kaszyk, Dong hyeon Park, Magnus Morton, Trevor Mudge, Murray Cole, Michael F P O'Boyle, Chaitali Chakrabarti, and Ronald Dreslinski. Cosparse: A software and hardware reconfigurable spmv framework for graph analytics. In *Proceedings of the 58th Design Automation Conference (DAC 2021)*. ACM Association for Computing Machinery, February 2021. 58th Design Automation Conference, DAC 2021 ; Conference date: 05-12-2021 Through 09-12-2021.

[20] M. Gao and C. 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, March 2016.

[21] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, and Martin Herbordt. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. 10 2020.

[22] 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.

[23] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. Understanding the performance of sparse matrix-vector multiplication. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP '08. IEEE, 2008.

[24] V. Govindaraju, C. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 503–514, Feb 2011.

[25] Werner H Greub. *Linear algebra*, volume 23. Springer Science & Business Media, 2012.

[26] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, September 1978.

[27] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[28] 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*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[29] 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, New York, NY, USA, 2007. ACM.

[30] Lihong Jia, Yonghong Gao, Jouni Isoaho, and Hannu Tenhunen. A new vlsi-oriented fft algorithm and implementation. In *Proceedings Eleventh Annual IEEE International ASIC Conference (Cat. No.98TH8372)*, pages 337–341, 1998.

[31] 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 *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[32] John H Kelm, Daniel R Johnson, Matthew R Johnson, Neal C Crago, William Tuohy, Aqeel Mahesri, Steven S Lumetta, Matthew I Frank, and Sanjay J 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.

[33] W. James Kent. BLAT - The BLAST-like alignment tool. *Genome Research*, 12(4):656–664, 2002.

[34] 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.

[35] 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, Washington, DC, USA, 2008. IEEE Computer Society.

[36] Sung Kim, Morteza Fayazi, Alhad Daftardar, Kuan-Yu Chen, Jielun Tan, Subhankar Pal, Tutu Ajayi, Yan Xiong, Trevor Mudge, Chaitali Chakrabarti, David Blaauw, Ronald Dreslinski, and Hun-Seok Kim. Versa: A dataflow-centric multiprocessor with 36 systolic arm cortex-m4f cores and a reconfigurable crossbar-memory hierarchy in 28nm. In *2021 Symposium on VLSI Circuits*, pages 1–2, 2021.

[37] Géraud P Krawezik and Gene Poole. Accelerating the ansys direct sparse solver with gpus. In *Symposium on Application Accelerators in High Performance Computing, SAAHPC*, 2009.

[38] 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.

[39] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nat Methods*, 9(4):357–359, 2012.

[40] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[41] Colin Yu Lin, Hayden Kwok-Hay So, and Philip HW Leong. A model for matrix multiplication performance on fpgas. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 305–310. IEEE, 2011.

[42] Colin Yu Lin, Ngai Wong, and Hayden Kwok-Hay So. Design space exploration for sparse matrix-matrix multiplication on fpgas. *International Journal of Circuit Theory and Applications*, 41(2):205–219, 2013.

[43] Guihua Liu and Quanyuan Feng. Asic design of low-power reconfigurable fft processor. In *2007 7th International Conference on ASIC*, pages 44–47, 2007.

[44] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC bioinformatics*, 14:117, 2013.

[45] 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, 2016.

[46] 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.

[47] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart Memories: A Modular Reconfigurable Architecture. *Computer Architecture, 2000. ISCA '00. Proceedings of the 27th International Symposium on*, pages 161–171, 2000.

[48] K. Matam, S. R. Krishna Bharadwaj Indarapu, and K. Kothapalli. Sparse matrix-matrix multiplication on modern architectures. In *2012 19th Int'l Conference on High Performance Computing*, pages 1–10, Dec 2012.

[49] Andrew Mccallum, Kamal Nigam, and Jason Rennie. Automating the construction of internet portals. 03 2000.

[50] Kenneth Moreland and Edward Angel. The fft on a gpu. pages 112–119, 01 2003.

[51] 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, New York, NY, USA, 2017. ACM.

[52] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. 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.

[53] Subhankar Pal, Siying Feng, Dong-hyeon Park, Sung Kim, Aporva Amarnath, Chi-Sheng Yang, Xin He, Jonathan Beaumont, Kyle May, Yan Xiong, et al. 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.

[54] 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 Taylor, Trevor Mudge, David Blaauw, Hun-Seok Kim, and Ronald 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 Technology*, pages C150–C151. IEEE, 2019.

[55] Dong-Hyeon Park, Jonathan Beaumont, and Trevor Mudge. Accelerating smith-waterman alignment workload with scalable vector computing. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 661–668, 2017.

[56] 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 Mudge, David 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. *IEEE Journal of Solid-State Circuits*, 55(4):933–944, 2020.

[57] Charles Phillips and John Parr. Signals, systems and transforms. 01 2003.

[58] 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 *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 389–402. IEEE, 2017.

[59] Karl Rupp. 42 years of microprocessor trend data. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/. Accessed: 2019-10-01.

[60] Yousef Saad. *Iterative methods for sparse linear systems*, volume 82. siam, 2003.

[61] Ahmed Saeed, M Elbably, G Abdelfadeel, and MI Eladawy. Efficient fpga implementation of fft/ifft processor. *International Journal of circuits, systems and signal processing*, 3(3):103–110, 2009.

[62] Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD Int'l conference on Management of data*, pages 979–990. ACM, 2014.

[63] S. Satpathy, K. Sewell, T. Manville, Y. Chen, R. Dreslinski, D. Sylvester, T. Mudge, and D. Blaauw. A 4.5tb/s 3.4tb/s/w 64×64 switch fabric with self-updating least-recently-granted priority and quality-of-service arbitration in 45nm cmos. In *2012 IEEE International Solid-State Circuits Conference*, pages 478–480, Feb 2012.

[64] Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. In *International Conference on Parallel Processing and Applied Mathematics*, pages 559–570. Springer, 2013.

[65] Aaron Smith. 6 new facts about Facebook, Feb 2014.

[66] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: 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, 2020.

[67] Shannon Steinfadt. SWAMP+: Enhanced Smith-Waterman Search for Parallel Models. *ICPPW*, 2012.

[68] Adam Szalkowski, Christian Ledergerber, Philipp Krähenbühl, and Christophe Dessimoz. SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. *BMC research notes*, 1(1):107, 2008.

[69] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor Mudge, and Ronald Dreslinski. Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 654–667, 2021.

[70] C. Tan, M. Karunaratne, T. Mitra, and L. 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, June 2018.

[71] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, Jae-Wook Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, March 2002.

[72] Robert A Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm.

[73] F V'zquez, G Ortega, JJ Fern'ndez, Inmaculada García, and Ester M Garzón. Fast sparse matrix matrix product based on ellr-t and gpu computing. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 669–674. IEEE, 2012.

[74] 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.

[75] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, Jan 2021.

[76] Ruochen Xu, Yiming Yang, Naoki Otani, and Yuexin Wu. Unsupervised cross-lingual transfer of word embedding spaces, 2018.

[77] Peiheng Zhang, Guangming Tan, and Guang R. Gao. Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform. *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications held in conjunction with SC07 - HPRCTA '07*, page 39, 2007.

[78] Mengyao Zhao, Wan Ping Lee, Erik P. Garrison, and Gabor T. Marth. SSW library: An SIMD Smith-Waterman C/C++ library for use in genomic applications. *PLoS ONE*, 2013.

[79] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018.