Transmuter: Bridging the Efficiency Gap using Memory and Dataflow Reconfiguration


*University of Michigan, USA  †Arizona State University, USA  ‡University of Edinburgh, UK
Email: subh@umich.edu

ABSTRACT

With the end of Dennard scaling and Moore’s law, it is becoming increasingly difficult to build hardware for emerging applications that meet power and performance targets, while remaining flexible and programmable for end users. This is particularly true for domains that have frequently changing algorithms and applications involving mixed sparse/dense data structures, such as those in machine learning and graph analytics. To overcome this, we present a flexible accelerator called Transmuter, in a novel effort to bridge the gap between General-Purpose Processors (GPPs) and Application-Specific Integrated Circuits (ASICs). Transmuter adapts to changing kernel characteristics, such as data reuse and control divergence, through the ability to reconfigure the on-chip memory type, resource sharing and dataflow at run-time within a short latency. This is facilitated by a fabric of light-weight cores connected to a network of reconfigurable caches and crossbars. Transmuter addresses a rapidly growing set of algorithms exhibiting dynamic data movement patterns, irregularity, and sparsity, while delivering GPU-like efficiencies for traditional dense applications. Finally, in order to support programmability and ease-of-adoption, we prototype a software stack composed of low-level runtime routines, and a high-level language library called TransPy, that cater to expert programmers and end-users, respectively.

Our evaluations with Transmuter demonstrate average throughput (energy-efficiency) improvements of 5.0× (18.4×) and 4.2× (4.0×) over a high-end CPU and GPU, respectively, across a diverse set of kernels predominant in graph analytics, scientific computing and machine learning. Transmuter achieves energy-efficiency gains averaging 4.2× and 2.0× over prior FPGA and CGRA implementations of the same kernels, while remaining on average within 9.3× of state-of-the-art ASICs.

CCS CONCEPTS

• Computer systems organization → Reconfigurable computing; Dataflow architectures; Multicore architectures.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PACT ’20, October 3–7, 2020, Virtual Event, GA, USA © 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8075-1/20/10 $15.00
https://doi.org/10.1145/3410463.3414627

KEYWORDS

Reconfigurable architectures, memory reconfiguration, dataflow reconfiguration, hardware acceleration, general-purpose acceleration

ACM Reference Format:

1 INTRODUCTION

The past decade has seen a surge in emerging applications that are composed of multiple kernels1 with varying data movement and reuse patterns, in domains such as Machine Learning (ML), linear algebra, and graph, image and signal processing. A growing number of such applications operate on compressed and irregular data structures [23, 50, 74], or on a combination of regular and irregular data [19, 22, 120]. Typical heterogeneous systems that accelerate these applications, such as those deployed in datacenters and High-Performance Computing (HPC) clusters, consist of CPUs paired with GPUs and other domain-specific Application-Specific Integrated Circuit (ASIC) based accelerators [40, 49, 76], or Field Programmable Gate Arrays (FPGAs) [88, 99, 115]. Coarse-Grained Reconfigurable Architectures (CGRAs) have also been proposed as promising alternatives for achieving near-ASIC performance [85, 108]. These platforms have been historically bounded by three conflicting constraints: programmability, algorithm-specificity, and performance/efficiency [73], as is illustrated in Fig. 1. Owing to these trade-offs, there is currently no single architecture that is the most efficient across a diverse set of workloads [92]. Thus, the rising complexity of modern applications and need for efficient computing necessitate a solution that incorporates:

• Flexibility. Ability to cater to multiple applications, as well as emerging applications with changing algorithms, that operate on both regular and irregular data structures.
• Reconfigurability. Enabling near-ASIC efficiencies by morphing the hardware to specific kernel characteristics, for applications that are composed of multiple cascaded kernels.

1This work refers to kernels as the building blocks of larger applications.
Programmability. Facilitating better adoption of non-GPP hardware by providing high-level software abstractions that are familiar to end-users and domain experts, and that mask the details of the underlying reconfigurable hardware.

To this end, we propose Transmuter, a reconfigurable accelerator that adapts to the nature of the kernel through a flexible fabric of light-weight cores, and reconfigurable memory and interconnect. Worker cores are grouped into tiles that are orchestrated by a control core. All cores support a standard ISA, thus allowing the hardware to be fully kernel-agnostic. Transmuter overcomes inefficiencies in vector processors such as GPUs for irregular applications [90] by employing a Multiple-Instruction, Multiple Data (MIMD) / Single-Program, Multiple Data (SPMD) paradigm. On-chip buffers and Scratch-Pad Memories (SPMs) are used for low-cost scheduling, synchronization and fast core-to-core data transfers. The cores interface to a High-Bandwidth Memory (HBM) through a two-level hierarchy of reconfigurable caches and crossbars.

Our approach fundamentally differs from existing solutions that employ gate-level reconfigurability (FPGAs) and core/pipeline-level reconfigurability (most CGRAs) — we configure the on-chip memory type, resource sharing, and dataflow, at a coarser granularity than contemporary CGRAs, while employing general-purpose cores as the compute units. Moreover, Transmuter’s reconfigurable hardware enables run-time reconfiguration within 10s of nanoseconds, faster than existing CGRA and FPGA solutions (Section 2.1).

We further integrate a prototype software stack to abstract the reconfigurable Transmuter hardware and support ease-of-adoption. The stack exposes two layers: (i) a C++ intrinsics layer that compiles directly for the hardware using a commercial off-the-shelf (COTS) compiler, and (ii) a drop-in replacement for existing high-level language (HLL) libraries in Python, called TransPy, that exposes optimized Transmuter kernel implementations to an end-user. Libraries are written by experts using the C++ intrinsics to access reconfigurable hardware elements. These libraries are then packaged and linked to existing HLL libraries, e.g. NumPy, SciPy, etc.

In summary, this paper makes the following contributions:

- **Demonstrates the flexibility of Transmuter** by mapping and analyzing 6 fundamental compute- and memory-bound kernels, that appear in multiple HPC and datacenter applications, onto 3 distinct Transmuter configurations.
- **Illustrates the significance of fast reconfiguration** by evaluating Transmuter on 10 end-to-end applications (one in detail) spanning the domains of ML and graph/signal processing, that involve reconfiguration at kernel boundaries.
- **Prototypes a compiler runtime and an HLL library called TransPy** that expose the Transmuter hardware to end-users through drop-in replacements for existing HLL libraries. The stack also comprises of C++ intrinsics, which foster expert programmers to co-design new algorithms.
- **Evaluates Transmuter against prior platforms** with two proposed designs, namely TransX1 and TransX8, that are each comparable in area to a high-end CPU and GPU.

In summary, Transmuter demonstrates average energy-efficiency gains of 18.4×, 4.0×, 4.2× and 2.0×, over a CPU, GPU, FPGAs and CGRAs respectively, and remains within 3.0×-32.1× of state-of-the-art ASICs. Fig. 1 (right) presents a summary of these comparisons.

## 2 BACKGROUND & MOTIVATION

In this section we provide background on traditional approaches and motivate the need for a reconfigurable design.

### 2.1 Contemporary Computing Platforms

ASICs have been a subject of extensive research in the dark silicon era for their superior efficiency [103]. However, ASICs compromise on generality by stripping away extraneous hardware, such as control logic, thus limiting their functionality to specific algorithms. An obvious solution is to design systems with multiple ASICs, but that leads to high under-utilization for applications with cascaded kernels. Moreover, fast-moving domains, such as ML, involve algorithms that evolve faster than the turnaround time to fabricate and test new ASICs, thus subjecting them to near-term obsolescence [17, 43]. Finally, ASICs are generally non-programmable, barring a few that use sophisticated software frameworks [2].

FPGAs have been successful for fast prototyping and deployment by eliminating non-recurring costs through programmable blocks and routing fabric. Moreover, high-level synthesis tools have reduced the low-level programmability challenges associated with deploying efficient FPGA-based designs [8, 62, 63]. Despite that, power and cost overheads prohibit FPGAs from adaptation in scenarios that demand the acceleration of a diverse set of kernels [16, 97, 98]. Besides, reconfiguration overheads of FPGAs are in the ms-μs range, even for partial reconfiguration [113, 117, 118], thus impeding run-time reconfiguration across kernel boundaries. CGRAs overcome some of the energy and performance inefficiencies of FPGAs by reconfiguring at a coarser granularity. However, CGRA reconfiguration usually happens at compile-time, and the few that support run-time reconfiguration allow only compute datapath reconfiguration [69], with overheads ranging from a few μs to 100s of ns [30, 33, 72]. Furthermore, many CGRAs require customized software stacks but have inadequate tool support, since they typically involve Domain-Specific Languages (DSLs) and custom ISAs [116].

### Table: Comparing platforms

<table>
<thead>
<tr>
<th>Platform</th>
<th>Programmability</th>
<th>Flexibility</th>
<th>Reconfig. Time</th>
<th>Memory-Bound (FLOPS/B)</th>
<th>Compute Bound (FLOPS/B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC</td>
<td>Low</td>
<td>Low</td>
<td>N.A.</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>CGRA</td>
<td>Low/Medium</td>
<td>Medium</td>
<td>O(I) μs, 100 ns</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>FPGA</td>
<td>Medium-High</td>
<td>High</td>
<td>O(I) μs, 100 ns</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>GPU</td>
<td>High</td>
<td>High</td>
<td>N.A.</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>CPU</td>
<td>High</td>
<td>High</td>
<td>O(10 nan)</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

---

Figure 1: Left: Transmuter compared to contemporary platforms in terms of software support, hardware flexibility and reconfiguration times. Right: Energy-efficiency comparisons for kernels spanning a wide range of arithmetic intensities (FLOPS/B). Note that for ASICs and CGRAs, no single hardware supports all kernels. Transmuter achieves 2.0× better average efficiency over state-of-the-art CGRAs, while retaining the programmability of GPPs.

---
Finally, while CPUs and GPUs carry significant energy and area overheads compared to lean ASIC designs, they are the de facto choice for programmers as they provide high flexibility and abstracted programming semantics [13]. Although GPUs are efficient across many regular (i.e., low control divergence) HPC applications, improving their effectiveness on irregular workloads remains a topic of research today [14, 21, 87].

### 2.2 Taming the Diversity across Kernels

Many real-world workloads consist of multiple kernels that exhibit differing data access patterns and computational intensities. In Fig. 2, we show the percentage execution times of key kernels that compose a set of 10 workloads in the domains of machine learning, graph analytics, and signal/image/video processing. These workloads are derived from an ongoing multi-university program to study software-defined hardware.

The underlying kernels exhibit a wide range of arithmetic intensities, from $10^2$ to $10^4$ of Floating-Point Operations per Byte, i.e., FLOPS/B (Fig. 1). We briefly introduce the kernels here. General (dense) Matrix-Matrix multiplication (GemM) and Matrix-Vector multiplication (GemV) are regular kernels in machine learning, data analytics and graphics [28, 32, 78]. Convolution is a critical component in image processing [4, 54] and convolutional neural networks [59]. Fast Fourier Transform (FFT) is widely used in speech and image processing [7, 82]. Sparse Matrix-Matrix multiplication (SpMM) is an important irregular kernel in graph analytics (GraphBLAS [53]), scientific computation [10, 25, 90, 119], and problems involving big data with sparse connections [45, 96]. Another common sparse operation is Sparse Matrix-Vector multiplication (SpMV), which is predominant in graph algorithms such as PageRank and Breadth-First Search [80], as well as ML-driven text analytics [6].

**Takeaways.** Fig. 2 illustrates that real-world applications exhibit diverse characteristics not only across domains, but also within an application. Thus, taming both the inter- and intra-application diversity efficiently is a significant piece of hardware calls for an architecture capable of tailoring itself to the characteristics of each composing kernel.

### 2.3 Hardware Support for Disparate Patterns

Intuition dictates that the diverse characteristics of kernels would demand an equivalent diversity in hardware. We study the implications of some key hardware choices here.

#### 2.3.1 On-Chip Memory Type: Cache vs. Scratchpad (SPM)

Cache and SPM are two well-known and extensively researched types of on-chip memory [9, 58, 112]. To explore their trade-offs, we performed experiments on a single-core system that employs these memories. We observed that:

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

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

#### 2.3.2 On-Chip Resource Sharing: Private vs. Shared

The performance of shared versus private on-chip resources is dependent on the working set sizes and overlaps across cores, i.e., inter-core data reuse. From our experiments we noted:

- When there is significant overlap between the threads’ working sets, sharing leads to speedups exceeding 10× over privatization. This is owed to memory access coalescing and deduplication of data in the shared mode.
- When cores work on disjoint data, there is insignificant difference in performance with sharing over no-sharing, if the union of the threads’ working sets fit on-chip.
- Regular kernels may exhibit strided accesses that can be hazardous for a shared multi-banked cache, due to conflicting accesses at the same bank.

#### 2.3.3 Dataflow: Demand-Driven vs. Spatial

We define demand-driven dataflow as that used by GPPs, wherein cores use on-demand loads/stores to read/write data and communicate through shared memory. In contrast, spatial dataflow architectures (e.g., systolic arrays) are data-parallel designs consisting of multiple Processing Elements (PEs) with direct PE-to-PE communication channels. Each PE receives data from its neighbor, performs an operation, and passes the result to its next neighbor [61]. If pipelined correctly, this form of data orchestration achieves the largest degree of parallelism. However, it is harder to map and write efficient software for certain applications on spatial architectures [48].

**Takeaways.** The on-chip memory type, resource sharing and dataflow are three key hardware design choices that are each amenable to a different workload characteristic. Thus, an architecture that reconfigures between these designs can accelerate workloads that exhibit a spectrum of characteristics.
On-Chip Memory

Resource Sharing

1D / 2D Systolic Sharing

---

Table 1: Reconfigurable features at each level in Transmuter. In the “hybrid” memory mode, banks are split between caches and SPMs.

<table>
<thead>
<tr>
<th>Dataflow</th>
<th>On-Chip Memory</th>
<th>Resource Sharing</th>
<th># Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand-driven</td>
<td>Cache / SPM / Hybrid</td>
<td>Private / Shared</td>
<td>6</td>
</tr>
<tr>
<td>Spatial</td>
<td>FIFO + SPM</td>
<td>1D / 2D Systolic Sharing</td>
<td>2</td>
</tr>
</tbody>
</table>

3 HIGH-LEVEL ARCHITECTURE

The takeaways from the previous section are the fundamental design principles behind our proposed architecture. Transmuter is a tiled architecture composed of a massively parallel fabric of simple cores. It has a two-level hierarchy of crossbars and on-chip memories that allows for fast reconfiguration of the on-chip memory type (cache/scratchpad/FIFO), resource sharing (shared/private) and dataflow (demand-driven/spatial). The various modes of operation are listed in Tab. 1. The two levels of memory hierarchy, i.e. L1 and L2, supports 8 modes each. Furthermore, each Transmuter tile can be configured independently, however such configurations are not evaluated in this work.

In this work, we identify three distinct Transmuter configurations to be well-suited for the evaluated kernels based on characterization studies on existing platforms (Sec. 2.2). These configurations are shown in Fig. 3 and discussed here.

- **Shared Cache (Trans-SC).** Trans-SC uses shared caches in the L1 and L2. The crossbars connect the cores to L1s and the tiles to L2s, respectively. This resembles a manycore system, but with a larger compute-to-cache ratio, and is efficient for regular accesses with high inter-core reuse.
- **Private Scratchpad (Trans-PS).** Trans-PS reconfigures the caches in L1 into SPMs while retaining the L2 as cache. The crossbars reconfigure to privatize the L1 (L2) SPMs to their corresponding cores (tiles). This configuration is suited for workloads with high intra-core but low inter-core reuse of data that is prone to cache-thrashing. The private L2 banks enable caching of secondary data, such as spill/fill variables.
- **Systolic Array (Trans-SA).** Trans-SA employs systolic connections between cores (or tiles), and is suited for highly data parallel applications where the work is relatively balanced between the cores. Transmuter supports both 1D and 2D systolic configurations. Note that the L2 is configured as a cache for the same reason as with Trans-PS.

While Transmuter supports a total of 64 configurations, we omit reporting a rigorous evaluation of other configurations given the space constraints of the paper. In the rest of the paper, we use the notation of \( N_T \times N_G \) Transmuter to describe a system with \( N_T \) tiles and \( N_G \) worker cores per tile.

4 HARDWARE DESIGN

A full Transmuter system is shown in Fig. 4 a). A Transmuter chip consists of one or more Transmuter (TM) clusters interfaced to High-Bandwidth Memory (HBM) stack(s) in a 2.5D configuration, similar to modern GPUs [67]. A small host processor sits within the chip to enable low-latency reconfiguration. It is interfaced to a separate DRAM module and data transfer is orchestrated through DMA controllers (not shown) [31]. The host is responsible for executing serial/latency-critical kernels, while parallelizable kernels are dispatched to Transmuter.

4.1 General-purpose Processing Element and Local Control Processor

A General-purpose Processing Element (GPE) is a tiny processor, with Floating-Point (FP) and Load/Store (LS) units, that uses a standard ISA. Its small footprint enables Transmuter to incorporate many such GPEs. The large number of GPEs coupled with Miss Status Holding Registers (MSHRs) in the cache hierarchy enables Transmuter to exploit Memory-Level Parallelism (MLP) across the sea of processors. The GPEs operate in a MIMD/SPMD fashion, and thus have private instruction (I-) caches.

GPEs are grouped into tiles and are coordinated by a small control processor, the Local Control Processor (LCP). Each LCP has private D- and ICaches that connect to the HBM interface. The LCP orchestrates GPEs using either static (e.g. greedy) or dynamic scheduling (e.g. skipping GPEs with full queues), thus trading-off code complexity for work balance.

4.2 Work and Status Queues

The LCP distributes work to the GPEs through private FIFO work queues. A GPE publishes its status via private status queues that interfaces to the LCP as shown in Fig. 4 c). The queues block access when there are structural hazards, i.e. if the queue is empty and a consumer attempts a POP, the consumer is idled until a producer PUSHes to the queue, thus circumventing wasted energy due to busy-waiting. This strategy is also used for systolic accesses, discussed next.

4.3 Reconfigurable Data Cache (R-DCache)

Transmuter has two layers of multi-banked memories, called Reconfigurable Data Caches (R-DCaches) (Fig. 4 – b, c). Each R-DCache bank is a standard cache module with enhancements to support the following modes:

- **CACHE.** Each bank is accessed as a non-blocking, write-back, write-no-allocate cache with a standard replacement policy. A cacheline physically resides in a single bank. Additionally, this
mode uses a simple stride prefetcher that boosts performance for regular workloads.

- **SPM.** The tag array, set-index logic, prefetcher and MSHRs are powered off and the bank is accessed as an SPM.
- **FIFO+SPM.** A partition of the bank is configured as SPM. The remainder is accessed as FIFO queues (Fig. 5 - left), using a set of head and tail pointers. The depth can be reconfigured through memory-mapped registers. The low-level abstractions for accessing the FIFOs using loads/stores is shown in Fig. 5 (right). This mode is used to implement spatial dataflow in Trans-SA (Fig. 3).

### 4.4 Reconfigurable Crossbar (R-XBar)

A multicasting $N_{src} \times N_{dst}$ crossbar creates one-to-one or one-to-many connections between $N_{src}$ source and $N_{dst}$ destination ports. Transmuter employs Swizzle-Switch Network (SSN)-based crossbars that support multicasting [47, 102]. These works and others [3] have shown that crossbars designs can scale better, up to radix-64, compared to other on-chip networks. We augment the crossbar design with a Crosspoint Control Unit (XCU) that enables reconfiguration by programming the crosspoints. A block diagram of a Reconfigurable Crossbar (R-XBar) is shown in Fig. 4. d). The R-XBars support the following modes of operation:

- **ARBITRATE.** Any source port can access any destination port, and contention accesses to the same port get serialized. Arbitration is done in a single cycle using a least-recently granted policy [102], but serialization latencies can reach up to $(N_{src} - 1)$ cycles. This mode is used in Trans-SC.
- **TRANSPARENT.** A requester can only access its corresponding resource, i.e. the crosspoints are set to 0/1 (Fig. 4. d). Thus, the R-XBar is transparent and incurs no arbitration or serialization in this mode. Trans-PS (L1/L2) and Trans-SA (L2) employ TRANSPARENT R-XBars.
- **ROTATE.** The R-XBar cycles through a set of one-to-one port connections programmed into the crosspoints. This mode also has no crossbar arbitration cost. Fig. 6 describes how the port-multiplexing is used to emulate spatial dataflow in a 1D systolic array configuration (Trans-SA).

There are two L1 R-XBars in a tile (Fig. 4 c). The upper R-XBar allows GPEs to access the L1 R-DCache, and the lower R-XBar amplifies on-chip bandwidth between the L1 and L2.

### 4.5 Synchronization

Transmuter implements synchronization and enforces happens-before ordering using two approaches. The first is implicit, in the form of work/status/R-DCache queue accesses that block when the queue is empty or full. Second, it also supports explicit synchronization through a global synchronization SPM for programs that require mutexes, condition variables, barriers, and semaphores. For instance, say that GPEs 0 and 1 are to execute a Critical Section (CS). With explicit synchronization, the programmer can instantiate a mutex in the synchronization SPM and protect the CS with it. The same can be achieved through implicit synchronization with the following sequence of events: 1) both GPEs → LCP, 2) LCP → GPE0, 3) GPE8 executes the CS, 4) GPE8 → LCP, 5) LCP → GPE1, 6) GPE1 executes the CS, 7) GPE1 → LCP, where → denotes POP-from and ← is PUSH-to the work or status queue.
does not implement hardware coherence, switching between certain Transmuter configurations entails cache flushes from L1 to L2, from L2 to HBM, or both. Levels that use the SPM/FIFO+SPM mode do not need flushing. Furthermore, our write-no-allocate caches circumvent flushing for streaming workloads that write output data only once. Even when cache flushes are inevitable, the overhead is small (<1% of execution time) for the evaluated kernels in Sec. 8.

5 KERNEL MAPPING

Transmuter is built using COTS cores that lend the architecture to be kernel-agnostic. Here, we present optimized mappings of the kernels in Sec. 2 on three configurations. Code snippets for three of our mappings are shown in Appendix C. Note that while executing memory-bound kernels, Transmuter powers-down resources within a tile to conserve energy.

5.1 Dense Matrix Multiplication and Convolution

GeMM. GeMM is a regular kernel that produces $O(N^3)$ FLOPS for $O(N^2)$ fetches and exhibits very high reuse [38]. It also presents contiguous accesses, thus showing amenability to a shared memory based architecture. Our implementation of GeMM on Trans-SC uses a common blocking optimization [70, 79]. We similarly implement GeMM on Trans-PS but with the blocked partial results stored in the private L1 SPMs. Naturally, Trans-PS misses the opportunity for data sharing. For Trans-SA, the GPEs execute GeMM in a systolic fashion with the $A$-rows streamed through the L2 cache, and the $B$-columns loaded from the L1 SPM.

GeMV. GeMV is a memory-bound kernel that involves lower FLOPS/B — $O(N^2)$ FLOPS for $O(N^2)$ fetches — than GeMM, but still involves contiguous memory accesses [29]. The Trans-SC and Trans-PS implementations are similar to those for GeMM, but blocking is not implemented due to lower data reuse. On Trans-SA, the vector is streamed into each GPE through the L2 cache, while the matrix elements are fetched from the L1 SPM. Each GPE performs a MAC and passes the partial sum and input matrix values to its neighbors. We avoided network deadlock in the GeMV and GeMM Trans-SA implementations by reconfiguring the FIFO depth of the L1 R-DCache (Sec. 4.3) to allow for sufficient buffering.

Conv. Conv in 2D produces $O(F^2 \cdot N^2 \cdot IC \cdot OC)$ FLOPS, for an $F \times F$ filter convolving with stride $S$ over an $N \times N$ image, with $IC$ input and $OC$ output channels. The filter is reused while computing one output channel, and across multiple images. Input reuse is limited to $O(F \cdot OC)$, for $S < F$. On Trans-SC, we assign each GPE to compute the output of multiple rows, to maximize the filter reuse across GPEs. For Trans-PS and Trans-SA, we statically partition each image into $B \times B \times IC$ sub-blocks, such that the input block and filter fit in the private L1 SPM. Each block is then mapped to a GPE for Trans-PS, and to a set of $F$ adjacent GPEs of a 1D systolic array for Trans-SA using a row stationary approach [18].

5.2 Fast Fourier Transform

FFT. FFT in 1D computes an $N$-point discrete Fourier transform in $\log(N)$ sequential stages. Each stage consists of $N/2$ butterfly operations. FFT applications often operate on streaming input samples, and thus are amenable to spatial dataflow architectures [26, 46].
Our Trans-SA mapping approach is similar to pipelined systolic ASICs; each stage is assigned to a single GPE, and each GPE immediately pushes its outputs to its neighbor. The butterflies in each stage are computed greedily. To reduce storage and increase parallelism, Trans-SA uses run-time twiddle coefficient generation when the transform size is too large for on-chip memory, e.g. >256 for 2x8, with the trade-off of making the problem compute-bound. On Trans-SC, the butterfly operations are distributed evenly among GPEs to compute a stage in parallel. LCPs assign inputs and collect outputs from GPEs. All cores synchronize after each stage. For Trans-PS, the same scheduling is used and partial results are stored in the L1 SPM.

5.3 Sparse Matrix Multiplication

SpMM. SpMM is a memory-bound kernel with low FLOPS that decreases with increasing sparsity, e.g. \( \sim 2N^3r^2 \), for uniform-random \( N \times N \) matrix with density \( r \). Furthermore, sparse storage formats lead to indirection and thus irregular memory accesses [66, 90]. We implement SpMM in Trans-SC using a prior outer product approach [90]. In the multiply phase of the algorithm, the GPEs multiply a column of \( A \) with the corresponding row of \( B \), such that the row elements are reused in the L1 cache. In the merge phase, a GPE merges all the partial products corresponding to one row of \( C \). Each GPE maintains a private list of sorted partial results and fills it with data fetched from off-chip. Trans-PS uses the same algorithm, but with the sorting list placed in private L1 SPM, given that SPMs are a better fit for operations on disjoint chunks in memory. Lastly, SpMM in Trans-SA is implemented following a recent work that uses sparse packing [60]. Both the columns of \( A \) and rows of \( B \) are packed in main memory. The computation is equally distributed across tiles.

SpMV. SpMV, similar to SpMM, is bandwidth-bound and produces low FLOPS (\( \sim 2N^3r \)) for a uniformly random \( N \times N \) matrix with density \( r \), and vector with density \( r_v \). We exploit the low memory traffic in the outer product algorithm for sparse vectors, mapping it to Trans-SC and Trans-PS. The GPEs and LCPs collaborate to merge the partial product columns in a thread fashion, with LCP 0 writing out the final elements to memory. SpMV on 1D Trans-SA is implemented using inner product on a packed sparse matrix as described in [41, 60]. The packing algorithm packs 64 rows as a slice, and assigns one slice to each 1x4 sub-tile within a tile. Each GPE loads the input vector elements into SPM, fetches the matrix element and performs MAC operations, with the partial results being streamed to its neighbor in the sub-tile.

For SpMM and SpMV, we use dynamic scheduling for work distribution to GPEs (Sec. 4.1) to exploit the amenability of sparse workloads to SPMD architectures [90].

6 PROTOTYPE SOFTWARE STACK

We implement a software stack for Transmuter in order to improve programmability and ease-of-adoption of our solution. The software stack has several components: a high-level Python API, and lower-level C++ APIs for the host, and Transmuter LCPs and GPEs. An outline of the software stack and a working Transmuter code example are shown in Fig. 7.

Figure 7: Transmuter software stack. Application code is written using Python and invokes library code for the host, LCPs and GPEs. The implementations are written by experts using our C++ intrinsics library. Also shown is an example of a correlation kernel on Trans-SA (host library code not shown). The end-user writes standard NumPy code and changes only the import package to transmuter-numpy (App.L1). Upon encountering the library call (App.L4), the host performs data transfers and starts execution on Transmuter. The LCP broadcasts the vector x to all GPEs (LCP.L7). Each GPE pops the value (GPE.L4), performs a MAC using its filter value (f) and east neighbor’s partial sum (GPE.L7), and sends its partial sum westward (GPE.E11). The last GPE stores the result into HB. The host returns to the application code after copying back the result y.

The highest level API, called TransPy, is a drop-in replacement for the well-known high-performance Python library NumPy, i.e. the Transpy API exactly mirrors that of NumPy. In the code example in Fig. 7, note that only one change is needed to convert the NumPy program to TransPy. The np.correlate function is trapped in TransPy, dispatched to the Transmuter host layer, and a pre-compiled kernel library is invoked. We use pybind11 as the abstraction layer between Python and C++. TransPy also contains Transmuter drop-in replacements for SciPy, PyTorch, NetworkX, and other libraries used in scientific computing, machine learning, graph analytics, etc.

TransPy invokes kernels that are implemented by library writers and expert programmers, with the aid of the C++ intrinsics layer. A Transmuter SPMD kernel implementation consists of three programs, one each for the host, LCP and GPE. The host code is written in the style of OpenCL, handling data transfers to and from Transmuter, launching computation, initializing reconfigurable parameters (e.g. R-DCache FIFO depth), and triggering reconfiguration if needed. On the Transmuter-side, notable API methods include those associated with the queue interface, for accessing SPMs and FIFOs, triggering cache flushes, and reconfiguration. Synchronization is handled using intrinsics that wrap around POSIX threads functions [83]. These calls enable synchronization at different granularities, such as globally, within tiles, and between LCPs. A set of these C++ intrinsics is listed in Appendix A, and the code example in Fig. 7 reflects the use of some of these calls.

Thus, the Transmuter software stack is designed to enable efficient use of the Transmuter hardware by end-users, without the burden of reconfiguration and other architectural considerations. At the same time, the C++ layer allows expert programmers to write
Table 2: Microarchitectural parameters of Transmuter gem5 model.

<table>
<thead>
<tr>
<th>Module</th>
<th>Microarchitectural Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPE/LCP</td>
<td>1-issue, 4-stage, in-order (Minor/GPU) core @ 1.6 GHz, tournament branch predictor, FUs: 2 integer (3 cycles), 1 integer multiply (3 cycles).</td>
</tr>
<tr>
<td>Work/StatusQueue</td>
<td>4 B, 4-entry FIFO buffer between each GPE and LCP within a tile, loads empty if empty and stores if full.</td>
</tr>
<tr>
<td>R-DCache (per bank)</td>
<td>L1 ICache: 4 KB, 4-way set-associative, 1-port, non-coherent cache with 8 MSHRs and 64 B block size, stride prefetcher of degree 2, word-granular (L1) / cacheline-granular (L2).</td>
</tr>
<tr>
<td>R-Xbar</td>
<td>Core: max. non-coherent crossbar with 1-cycle response, 1-cycle arbitration latency, 0 to (Nl_unx-1) serialization latency depending upon number of conflicts.</td>
</tr>
<tr>
<td>GPE/LCP</td>
<td>4 KB, 4-way set-associative, 1-port, non-coherent cache with 8 MSHRs and 64 B block size.</td>
</tr>
<tr>
<td>Sync. SPM</td>
<td>4 KB, 1-port, physically-addressed scratchpad.</td>
</tr>
<tr>
<td>Main Memory</td>
<td>1 HBM2 stack: 16 64-bit pseudo-channel, each @ 8000 MB/s, 80-150 ns average access latency.</td>
</tr>
</tbody>
</table>

Table 3: Specifications of baseline platforms and libraries evaluated.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Specifications</th>
<th>Library Name and Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel i7-6800K, 4 cores/8 threads at 4.0-4.6 GHz, 16 GB DDR3 memory @ 1866 MHz, AXI4, SERR2.4, 122 mm² (14 nm)</td>
<td>MEL 2018.3.221 (GeMm/Gem5/Spanish)</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Tesla V100, NVIDIA CUDA cores at 1.25 GHz, 16 GB HBM2 memory at 900 GB/s, 815 mm² (12 nm)</td>
<td>cuBLAS v10.1 (GeMm/GeMm/Spanish), cuDNN v7.4.3 (Conv), cuFFT v10.0 (FFT), CUSP v5.5.1 (Span), cuSPARSE v8.0 (Span)</td>
</tr>
</tbody>
</table>

While retaining the gem5 model for the rest of the system. Offline traces are generated on a native machine and streamed through these engines. This allows us to simulate Transmuter system sizes up to one 64x64 cluster. The trace-driven model is pessimistic to an average of 4.5% over the gem5 core-based model. To simulate multiple Transmuter clusters, we build analytical models from gem5-derived bandwidth and throughput scaling data (Sec. 8.2).

We implemented each kernel in C++ and hand-optimized it for each of the Transmuter configurations using the intrinsics calls in Sec. 6. Code compilation was done using an Arm GNU compiler with the -O2 flag. All experiments were done using single-precision floating-point (FP) arithmetic.

7.2 Power and Area Models

We designed RTL models for Transmuter hardware blocks and synthesized them. The GPEs and LCPs are built as Arm Cortex-M4Fs. For the R-Xbar, we use the SSN design proposed in [102] augmented with an XCU. The R-DCaches are cache modules enhanced with SPM/FIFO control logic.

The crossbar and core power models are based on RTL synthesis reports and the M4F specification [1]. The R-Xbar power model is calibrated against the data reported in [102]. For the caches and synchronization SPM, we used CACTI 7.0 [84] to estimate the dynamic energy and leakage power. We further verified our power estimate for SpAM on Transmuter against a prior SpAM ASIC prototype [91], and obtained a pessimistic deviation of 17% after accounting for the architectural differences. Finally, the area model uses estimates from synthesized Transmuter blocks.

We note that this work considers only the chip power on all platforms for fair comparisons. We used standard profiling tools for the CPU and GPU, namely nvprof and RAPL. For the GPU, we estimated the HBM power based on per-access energy [89] and measured memory bandwidth, and subtracted it out. The power is scaled for iso-technology comparisons using quadratic scaling.

8 EVALUATION

We evaluate the configurations in Sec. 3 on the kernels in Sec. 5. We then compare the best-performing Transmuter to the CPU and GPU, and present a case study of an application that exercises rapid reconfiguration. Lastly, we present comparisons with prior platforms and power/area analysis. Other kernels in the domain of linear algebra have been evaluated for different resource sharing configurations on a preliminary version of Transmuter [104].

8.1 Performance with Different Configurations

Fig. 8 presents performance comparisons between Trans-SC, Trans-PS and Trans-SA. This analysis was done on a small 2x8 system to stress the hardware. The results show that the best performing Transmuter configuration is kernel-dependent, and in certain cases also input-dependent. Fig. 9 shows the cycle breakdowns and work imbalance across GPEs.

For GeMm, Trans-SC achieves high hit rates (>99%) as efficient blocking leads to good data reuse. Trans-PS suffers from capacity misses due to lack of sharing, noted from the large fraction of L2 misses. Further, Trans-SC performs consistently better than Trans-SA, as it does not incur the overhead of manually fetching data into
A shared cache configuration outperforms others for GeMM, GeMV and Conv due to sufficient data sharing and reuse.
by a 6.4× increase in power. Other parameters such as work/status queue depths were chosen such that GPEs never idled waiting on the LCP. The optimal number of tiles, GPEs per tile, and memory channels depend on the final suite of applications that will be mapped onto a future Transmuter prototype.

8.4 Performance with Varying Control Divergence and Data Reuse

In Section 2.2, we characterized some fundamental kernels based on their control divergence, data reuse and arithmetic intensity. We now build an intuition about the architectural advantages of Transmuter over a GPU for applications with notable contrast in these characteristics. We implement a parallel microbenchmark on Transmuter and the GPU that allows independently tuning the divergence and reuse. Fig. 11 (left) illustrates this application. The reuse (R) is controlled by the size of the coefficient array, while divergence (D) scales with the number of bins since threads processing each input element apply functions unique to a bin.

While this is a synthetic application, it is representative of real-world algorithms that perform image compression using quantization. We execute it with a batch of 1,000 32×32 images on 4×16 Transmuter, and compare it with the GPU running 64 threads (2 warps, inputs in shared memory) to ensure fairness. Fig. 11 (right) presents two observations:

- The performance of Transmuter roughly doubles as the number of divergent paths double. This is because threads executing different basic blocks get serialized in the SIMT paradigm (as they are in the same warp), whereas they can run asynchronously in Transmuter’s SPMD design.
- Transmuter has the inherent flexibility to reconfigure based on the input size. In this example, Trans-PS is best-performing until R=4. Beyond that, switching to Trans-SC enables better performance – up to 7.4× over Trans-PS – as the benefit of sharing the coefficient array across GPEs in Trans-SC outweighs the higher latency R-Xbar accesses.

Takeaways. Transmuter’s SPMD paradigm naturally lends itself well to kernels involving large control divergence, and its ability to reconfigure dynamically allows it to perform well for very low- and high-reuse, as well as mixed-reuse, workloads.

8.5 Comparison with the CPU and GPU

Next, we compare the best-performing Transmuter configuration with the CPU and GPU that run optimized libraries (Tab. 3). The throughput and energy-efficiency gains of Transmuter for each kernel in Sec. 5 are presented in Fig. 12. We compare TransX1 to the CPU and TransX8 to the GPU.

**Compute-Bound Kernels (GeMV, Conv, FFT).** TransX1 harnesses data parallelism leading to a performance improvements of 1.2-2.5× over the CPU, despite clocking at ¼ the speed of the deeply-pipelined CPU cores. The benefit of Transmuter’s simple cores and efficient crossbars appear in the form of energy-efficiency gains, ranging from 3.3-20.6×, owed largely to the power consumption of bulky out-of-order CPU cores. Over the GPU, TransX8 gets performance gains of 1.3-2.6× and efficiency improvements of 0.8-5.8× with an efficient implementation on Trans-SC for GeMV and Conv. The ~20% energy-efficiency loss for GeMV is explained by the amenability of GeMV to a SIMT paradigm; although the performance is similar between SIMT and SPMD, SPMD incurs slightly larger energy costs associated with higher control overhead over SIMT. On FFT, Transmuter sustains consistent performance scaling using systolic dataflow of Trans-SA, with each tile operating on an independent input stream leading to minimum conflicts. The gap between throughput gain (4.6×) and energy-efficiency gain (1.4×) over the GPU is explained by the cuFFT algorithm that is more efficient for batched FFTs.

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

On SpMV and SpMV, the performance of Transmuter is highly sensitive to the densities and sizes of the inputs, with improvements ranging from 4.1-110.8× over the CPU and 5.1-17.0× over the GPU. Switching between Trans-SC and Trans-PS enables SpMV to overcome the CPU’s limitation of an inflexible cache hierarchy, as well as harnesses high MLP across the sea of GPEs. While Transmuter is memory-bottlenecked for SpMV, SpMV is bounded by the scheduling granularity of packing algorithm deployed on Trans-SA. Despite that, for SpMV, TransX1 outperforms both the CPU as well as the GPU that has 7.2× greater available bandwidth. In case of the GPU, while there are sufficient threads to saturate the SMS, the thread divergence in the SIMT model is the bottleneck. The GPU achieves
Table 4: Estimated speedups for the end-to-end workloads in Fig. 2.

<table>
<thead>
<tr>
<th>Workload</th>
<th>DANMF</th>
<th>LSTM</th>
<th>Marian</th>
<th>MaxCut</th>
<th>MFCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransX1 over CPU</td>
<td>4.1x</td>
<td>1.1x</td>
<td>2.2x</td>
<td>6.2x</td>
<td>1.7x</td>
</tr>
<tr>
<td>TransX8 over GPU</td>
<td>3.5x</td>
<td>3.8x</td>
<td>2.1x</td>
<td>7.2x</td>
<td>1.6x</td>
</tr>
</tbody>
</table>

just 0.6% and 0.002% of its peak performance, respectively for SpMM and SpMV, impaired by memory and synchronization stalls. In comparison, SPMD on Transmuter reduces synchronization, resulting in 21-42% time spent on useful computation (Fig. 9). For SpMM, outer product on Trans-SC and Trans-PS enables ASIC-level performance gains of 5.9-11.6× [90] over the GPU as it minimizes off-chip traffic and exploits asynchronicity between GPEs. As with GeMV, disabling bandwidth-starved resources contributes to energy-efficiency gains.

**Effect of Iso-CPU Bandwidth.** TransX1 uses one HBM stack that provides 125 GB/s peak bandwidth, about 3.6× greater than the DDR3 bandwidth to the CPU. If given this lower bandwidth of 17.4× and 3.1× for SpMM and SpMV, respectively. For GeMV, TransX1 remains within a modest 6% of the CPU with this low bandwidth.

8.6 End-to-End Workload Analysis

We report the estimated speedups of Transmuter over the CPU and GPU for the end-to-end workloads (Fig. 2) in Tab. 4. File I/O and cross-platform data transfer times are excluded for all platforms. Transmuter achieves speedups averaging 3.1× over the CPU and 3.2× over the GPU.

Next, we elucidate how rapid reconfiguration enables efficient execution of workloads that involve mixed sparse/dense computation in an inner loop. We make a case study on a representative mixed-data application, namely Sinkhorn, that performs iterative computation to determine the similarity between documents [64, 100]. Sinkhorn computation typically involves large, sparse matrices in conjunction with dense matrices. We implement the algorithm described in [19] (see Appendix B). The inner loop has two major kernels: a GeMV operation masked by a sparse weight matrix (M-GeMV), and a dense matrix - sparse matrix multiplication (DMSpM).

The mapping on Transmuter is shown in Fig. 13. M-GeMV uses a variation of blocked-GeMV, wherein only rows/columns of the dense matrices that generate an element with indices corresponding to non-zeros in the weight matrix are fetched and multiplied. DMSpM uses a simplified outer product algorithm similar to SpMM (Sec. 5.3) that splits the kernel into DMSpM-Multiply and DMSpM-Merge.

We show the analysis of Sinkhorn on different Transmuter sizes in Fig. 14. As observed, M-GeMV and DMSpM-Multiply exhibit the best performance in Trans-SC configuration, due to good data reuse across GPEs. In contrast, DMSpM-Merge has optimal performance on Trans-PS, exhibiting a 1.8-2.0× improvement over Trans-SC. Therefore, the optimal Sinkhorn mapping involves two reconfigurations per iteration: Trans-SC → Trans-PS before the start of DMSpM-Merge, and Trans-PS → Trans-SC at the end of it, for the next M-GeMV iteration. Recall from Sec. 4.7 that the reconfiguration time is <10 cycles and hence does not perceptibly impact the performance or energy. Cache flushing (net 0.2% of the total execution time) is required for M-GeMV but not DMSpM, as DMSpM uses a streaming algorithm. Overall, dynamic reconfiguration results in 47.2% and 95.9% better performance and Energy-Delay Product (EDP), respectively, over Trans-SC-only for the 4x16 Transmuter. A heterogeneous solution is also compared against, where M-GeMV is done on the CPU and DMSpM on the GPU, but this implementation is bottlenecked by CPU → GPU data transfers. As seen from Fig. 14, the 4x16 Transmuter achieves 40.1× and 149.3× lower EDP than the GPU and heterogeneous solutions, respectively.

8.7 Comparison with Other Platforms

Tab. 5 presents the estimated energy-efficiency improvements of Transmuter over recent FPGA, CGRA and ASIC designs. The efficiencies reported in prior work are scaled quadratically for iso-technology comparisons. Transmuter achieves average efficiency gains of 4.2× and 2.0× over FPGAs and CGRAs, respectively, and is within 9.3× (maximum 32.1×) of state-of-the-art ASICs for the evaluated kernels.

8.8 Power and Area

Tab. 6 details the power consumption and area footprint of a 64×64 Transmuter cluster in 14 nm. Most of power is consumed by the network and memory, i.e. L1 R-XBars, R-DCaches and ICaches, while the cores only consume 20.8%. This is consistent with a growing awareness that the cost of computing has become cheaper than the cost to move data, even on-chip [42]. GPEs and L1 R-XBars, the
most frequently switched modules, consume 84.2% of the total dynamic power. The estimated power for a single Transmuter cluster is 13.3 W in 14 nm with an area footprint within 1.5% of the CPU, with an estimated worst-case reconfiguration overhead of 74.9 nJ.

9 RELATED WORK

A plethora of prior work has gone into building programmable and reconfigurable systems in attempts to bridge the flexibility-efficiency gap. A qualitative comparison of our work over related designs is shown in Tab. 7. Transmuter differentiates by supporting two dataflows, reconfiguring faster at a coarser granularity, and supporting a COTS ISA/compiler.

Reconfigurability. A few prior works reconfigure at the sub-core level [20, 44, 55, 77, 98] and the network-level [37, 56, 86, 109]. In contrast, Transmuter uses native in-order cores and the reconfigurability lies in the memory and interconnect. Some recent work propose reconfiguration at a coarser granularity [5, 20, 71, 98], PipeRench [36] builds an efficient reconfigurable fabric and uses a custom compiler to map a large logic configuration on a small piece of hardware. HRL [33] is an architecture for near-data processing, which combines coarse- and fine-grained reconfigurable blocks into a compute fabric. The Raw Microprocessor [109] implements a tiled architecture focusing on developing an efficient, distributed interconnect. Stream Dataflow [86] and SPU [20] reconfigure at runtime, albeit with non-trivial overheads to initialize the Data-Flow Graph (DFG) configuration. Transmuter, on the other hand, relies on flexible memories and interconnect that enable fast on-the-fly reconfiguration catering to the data needs of the application.

Flexibility. Prior work has also delved into efficient execution across a wide range of applications. Plasticine [98] is a reconfigurable accelerator for parallel patterns, consisting of a network of Pattern Compute/Memory Units that can be configured at compile-time to suit an algorithm. Stream Dataflow [86] is a new computing model, that efficiently executes algorithms that are expressible as DFGs, with the inputs/outputs specified as streams with a particular access pattern. SPU [20] targets data-dependence using a novel dataflow control model on a reconfigurable systolic compute-fabric. The flexibility of Transmuter stems from the use of general-purpose cores as well as the reconfigurable memory subsystem that morphs the dataflow and on-chip memory, thus catering to both inter- and intra-workload diversity.

Programmability. There have been proposals for programmable CGRAs that abstract the low-level hardware. Some work developed custom programming models, such as Rigel [52] and MaPU [114]. Others extend an existing ISA to support their architecture, such as Stitch [107] and LACore [105]. Plasticine [98] uses a DSL like Spatial [57]. Ambric [39] is a commercial system composed of asynchronous cores with a software stack that automatically maps Java code onto the processor-array. Transmuter distinguishes itself by using a standard ISA supported by a simple library of high-level language intrinsics and a COTS compiler, thus alleviating the need for ISA extensions or a DSL.

10 CONCLUSION

This work tackled the important challenge of bridging the flexibility-efficiency gap with Transmuter. Transmuter consists of simple processors connected to a network of reconfigurable caches and crossbars, that support fast reconfiguration of the memory, resource sharing and dataflow, thus tailoring Transmuter to the workload characteristics. We also presented a software stack comprised of drop-in replacements for standard Python libraries. We demonstrated Transmuter’s performance and efficiency on a suite of fundamental kernels, as well as mixed data-based multi-kernel applications. Our evaluation showed average energy-efficiency improvements of 46.8× (9.8×) over the CPU (GPU) for memory-bound kernels and 7.2× (1.6×) for compute-bound kernels. In comparison to state-of-the-art ASICs that implement the same kernels, Transmuter achieves average energy-efficiencies within 9.3×.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. The material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7864. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.
REFERENCES


A. LOW-LEVEL PROGRAMMING INTERFACE

Table 8: Critical host- and Transmutor-side C++ intrinsics used to write optimized kernel libraries (TID = Tile ID, GID = GPE ID).

<table>
<thead>
<tr>
<th>Host Intrinsic Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_LAUNCH()</td>
<td>Trigger Transmutor to start</td>
</tr>
<tr>
<td>C_JOINT()</td>
<td>Block until all execution threads finish</td>
</tr>
<tr>
<td>R_SEND_DATA()</td>
<td>Mem-copy from host to Transmutor</td>
</tr>
<tr>
<td>R_RET_DATA()</td>
<td>Mem-copy from Transmutor to host</td>
</tr>
<tr>
<td>R_SET_ARG()</td>
<td>Pass a pointer to an argument to an LCP or GPE</td>
</tr>
<tr>
<td>R_GET_Y()</td>
<td>Dynamically compile GPE's LCP code</td>
</tr>
<tr>
<td>R_SYNC_ALL()</td>
<td>Synchronize with all LCPs, GPEs, and work queue of a free GPE</td>
</tr>
<tr>
<td>R_JOIN()</td>
<td>Trigger GPE's/NCS reconfiguration</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transmutor Intrinsic Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_D_READ(addr)</td>
<td>Read a word from SPX</td>
</tr>
<tr>
<td>T_ST_WRITE(addr, val)</td>
<td>Write a word into SPX</td>
</tr>
<tr>
<td>T_SA_PUSH(DIR, val)</td>
<td>Pop data from systolic neighbor GPE</td>
</tr>
<tr>
<td>T_S_P_PUSH(val, GID)</td>
<td>Push data to systolic neighbor GPE</td>
</tr>
<tr>
<td>T_NC_PUSH(GID)</td>
<td>Pop data from work status queue</td>
</tr>
<tr>
<td>T_SHARED_WRITE<a href="">S2</a></td>
<td>Pushes a word to the work queue of a free GPE</td>
</tr>
<tr>
<td>T_WorkQ_PUSH_P(vide)</td>
<td>Broadcasts to the work queues of all GPEs in the tile</td>
</tr>
<tr>
<td>T_FLUSH_LEVELS(S2)()</td>
<td>Merges dirty data from to the next level</td>
</tr>
<tr>
<td>T_SPM_R(LEVEL)()</td>
<td>Get a pointer to the bottom of D-Cache SPM</td>
</tr>
<tr>
<td>T_SPM_WRITE(LEVEL)()</td>
<td>Get a pointer to the top of D-Cache SPM</td>
</tr>
<tr>
<td>T_SYNC_LCPs()</td>
<td>Synchronize with all LCPs</td>
</tr>
<tr>
<td>T_SYNC_TILE()</td>
<td>Synchronize with all GPEs and LCPs in the tile</td>
</tr>
<tr>
<td>T_SYNC_ALL()</td>
<td>Synchronize with all LCPs, GPEs, and work queue of a free GPE</td>
</tr>
<tr>
<td>T_FLUSH_LEVELS(S2)()</td>
<td>Put self into sleep to conserve power</td>
</tr>
<tr>
<td>T_RECONF_MEMLEVELS(S2)</td>
<td>Self-reconfigures R-D-Cache/NCS for wait for host</td>
</tr>
</tbody>
</table>

B. SINKHORN ALGORITHM

Algorithm B.1 Sinkhorn Distance (MATLAB syntax)

```matlab
function Sinkhorn(query, data, M, y, e)
    a = size(M, 2);  % M: distance matrix, y: regularization parameter, e: tolerance
    H = ones(length(query), a) / length(query);  % Masked-GeMM
    K = exp(-M/y);  % DMSpM
    err = oo;  % \|U\|_\infty = \sum_i |U_i| |
    while err > e do
        V = data(/K^T);  % \|U\|_2 = \sum_i |U_i|
        err = \|U - U_{prev}\|^2 / \|U\|^2;  % \|U\|_2 = \sum_i |U_i|^2
        end while
        return \|D\|_2 = \sum_i |D_{ij}|;  % Sinkhorn distances between query and data
end function
```

C. SELECTED KERNEL IMPLEMENTATIONS

Algorithm C.1 GeMV on Transmutor in Trans-SC configuration

```matlab
function GeMV_LCP(A, B, C, N, a, b)
    C = @A'B + b'C;  % N: matrix size; N, GPE: number of GPEs in a tile
    for row = 0 to N - 1 do
        T_WORKQ_PUSH(GID, row);
        GID = (GID == N_GPE - 1) ? 0 : (GID + 1);
    end for
end function
```

Algorithm C.2 SpMV on Transmutor in Trans-SA configuration

```matlab
function SpMV_LCP(A_rowID, A_colID, A, B, B_partition, C, N, P)
    C = A'B;  % N, GPE: number of GPEs in a tile
    T_WORKQ_PUSH_P(vide);
    for row = 0 to N - 1 do
        if C[row] > 0 then
            T_ST_PUSH(A[row][col] * B[col]);  % T_ST_PUSH: push to work queue of a free GPE
            partial_sum += A[row][col] * B[col];
        end for
    end while
end function
```

Algorithm C.3 FFT on Transmuter in Trans-SA configuration

function $\text{FFT}_\text{LCP}$ (input, output, N, INPUT_LCP, OUTPUT_LCP)
- N: FFT size, $\log_2 N$: FFT stages, S: Step size, P: Next step size
  if INPUT_LCP then
    for $i \leftarrow 0$ to $N - 1$ do
      T_WORKQ_PUSH(0, input[i]);
    end for
  end if
  if OUTPUT_LCP then
    for $i \leftarrow 0$ to $N - 1$ do
      output[i] = T_STATUSQ_POP($\log_2 N - 1$);
    end for
  end if
end function

function $\text{FFT}_\text{GPE}$ (input, output, N, S, P)
  $ID = \text{GBD} + \text{tileID} \times N_{GPE}$;
  $sp = T_{SPM\_BOT<n>}(i)$;
  for $i \leftarrow 0$ to $N/2$ do
    $in_1 = (ID = 0) ? T_{WORKQ\_POP()} : T_{SA\_POP}(\text{Dir} :: \text{West})$;
    $in_2 = (ID = 0) ? T_{WORKQ\_POP()} : T_{SA\_POP}(\text{Dir} :: \text{West})$;
    output, output = Butterfly($in_1$, $in_2$);
    if $ID = \log_2 N - 1$ then
      T_STATUSQ_PUSH(output);
      T_STATUSQ_PUSH(output);
    else
      T_STATUSQ_PUSH(sp + i, output);
      T_STATUSQ_PUSH(sp + S + i, output);
      if $i > P - 1$ then
        T_SA_PUSH(\text{Dir} :: \text{East}, T_LD_WORD(sp + i + P));
        T_SA_PUSH(\text{Dir} :: \text{East}, output);
      end if
    end if
  end for
  T_SA_PUSH(\text{Dir} :: \text{East}, T_LD_WORD(sp + S + i));
  T_SA_PUSH(\text{Dir} :: \text{East}, T_LD_WORD(sp + S + P + i));
end function