Transmuter: Bridging the Efficiency Gap using Memory and Dataflow Reconfiguration

Transmuter: Bridging the Efficiency Gap using Memory and Dataflow Reconfiguration

Subhankar Pal*, Siying Feng*, Dong-hyeon Park*, Sung Kim*, Aporva Amarnath*, Chi-Sheng Yang*, Xin He*, Jonathan Beaumont*, Kyle May*, Yan Xiong†, Kuba Kaszyk‡, John Magnus Morton‡, Jiawen Sun‡, Michael O'Boyle‡, Murray Cole‡, Chaitali Chakrabarti†, David Blaauw*, Hun-Seok Kim*, Trevor Mudge*, Ronald Dreslinski*

*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 3.4× 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**; *Data flow architectures*; Multicore architectures.

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

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:
Subhankar Pal, Siying Feng, Dong-hyeon Park, Sung Kim, Aporva Amarnath, Chi-Sheng Yang, Xin He, Jonathan Beaumont, Kyle May, Yan Xiong, Kuba Kaszyk, John Magnus Morton, Jiawen Sun, Michael O'Boyle, Murray Cole, Chaitali Chakrabarti, David Blaauw, Hun-Seok Kim, Trevor Mudge, and Ronald Dreslinski. 2020. Transmuter: Bridging the Efficiency Gap using Memory and Dataflow Reconfiguration. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3410463.3414627

1 INTRODUCTION

The past decade has seen a surge in emerging applications that are composed of multiple kernels[1] with varying data movement and reuse patterns, in domains such as machine learning (ML), and graph, image and signal processing. A growing number of such applications operate on compressed and *irregular* data structures [22, 51, 73], or on a combination of regular and irregular data [18, 21, 118]. While conventional CPU-GPU systems generally suffice for desktop computing [28], arenas such as high-performance computing (HPC) clusters and datacenters that demand higher performance for such applications require more specialized hardware; such systems are typically comprised of CPUs paired with GPUs and other domain-specific application-specific integrated circuit (ASIC) based accelerators [40, 50, 75], or field programmable gate arrays (FPGAs) [85, 96, 113]. coarse-grained reconfigurable architectures (CGRAs) have also been proposed as promising alternatives for achieving near-ASIC performance [82, 106]. These platforms have been historically bounded by three conflicting constraints: *programmability*, *algorithm-specificity*, and *performance/efficiency* [72], 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 [89].

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.

[1] This work refers to *kernels* as the building blocks of larger applications.

Platform	Programmability	Flexibility	Reconfig. Time
ASIC	Low/None	Low	N.A.
CGRA	Low-Medium	Medium	~1 µs - ~100 ns
FPGA	Medium	Medium-High	~1 ms - ~1 µs
GPU	High	High	N.A.
CPU	High	High	N.A.
Transformer	High	High	~10 ns

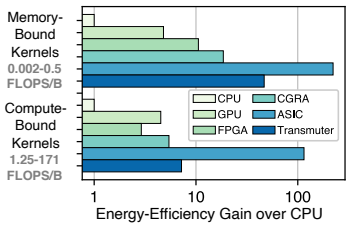

Figure 1: ***Left:*** **Transmuter compared to contemporary platforms in terms of programmability, hardware flexibility and reconfiguration overhead.** ***Right.*** **Energy-efficiency comparisons for kernels spanning a wide range of arithmetic intensities (FLOPS/B). Note that for ASICs and CGRAs, no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

- **Reconfigurability.** Enabling near-ASIC efficiencies by morphing the hardware to specific kernel characteristics, for applications that are composed of multiple cascaded kernels.
- **Programmability.** Facilitating better adoption of non-GPP hardware by providing high-level software abstractions that are familiar to end-users and domain experts, and that mask the details of the underlying reconfigurable hardware.

To this end, we propose `Transmuter`, a reconfigurable accelerator that adapts to the nature of the kernel through a flexible fabric of light-weight cores, and reconfigurable memory and interconnect. Worker cores are grouped into tiles that are each orchestrated by a control core. All cores support a standard ISA, thus allowing the hardware to be fully *kernel-agnostic*. `Transmuter` overcomes inefficiencies in vector processors such as GPUs for irregular applications [87] by employing a multiple-instruction, multiple data (MIMD) / single-program, multiple data (SPMD) paradigm. On-chip buffers and scratchpad 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 reconfigure the on-chip *memory type*, *resource sharing*, and *dataflow*, at a coarser granularity than contemporary CGRAs, while employing general-purpose cores as the compute units. Moreover, `Transmuter`'s reconfigurable hardware enables run-time reconfiguration within 10s of nanoseconds, faster than existing CGRA and FPGA solutions (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:

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

In summary, `Transmuter` demonstrates average energy-efficiency gains of 18.4×, 4.0×, 3.4× 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 AND MOTIVATION

In this section, we provide a background on conventional computing platforms, and motivate the need for a reconfigurable design.

2.1 Contemporary Computing Platforms

ASICs have been the subject of extensive research in the dark silicon era due to their superior performance and efficiency [100]. 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, despite efforts on accelerating the design flow [20], thus subjecting them to near-term obsolescence [16, 43]. Finally, ASICs are generally non-programmable, barring a few that use sophisticated software frameworks [1].

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 [7, 61, 62]. Despite that, power and cost overheads prohibit FPGAs from adaptation in scenarios that demand the acceleration of a diverse set of kernels [15, 94, 95]. Besides, reconfiguration overheads of FPGAs are in the ms-µs range, even for partial reconfiguration [111, 115, 116], thus impeding fast run-time reconfiguration across kernel boundaries. CGRAs overcome some of the energy and performance inefficiencies of FPGAs by reconfiguring at a coarser granularity. However, CGRA reconfiguration usually happens at compile-time, and the few that support run-time reconfiguration only support compute datapath reconfiguration [68], with overheads ranging from a few µs to 100s of

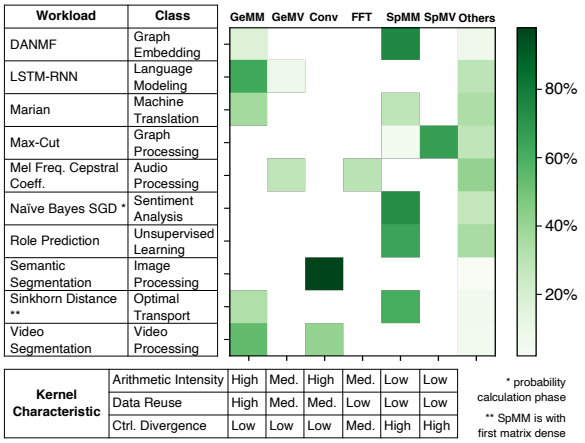

Figure 2: Fraction of execution time of kernels in applications spanning the domains of ML, signal processing, and graph analytics [14, 18, 21, 22, 51, 67, 73, 74, 78, 118] on a heterogeneous CPU-GPU platform. Some key characteristics, namely arithmetic intensity, data reuse and divergence, of each kernel are also listed.

ns [30, 33, 71]. Furthermore, many CGRAs require customized software stacks but have inadequate tool support, since they typically involve domain-specific languages (DSLs) and custom ISAs [114].

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 [12]. Although GPUs are efficient across many regular HPC applications, *i.e.* those exhibiting low control divergence, improving their effectiveness on irregular workloads remains a topic of research today [13, 84].

2.2 Taming the Diversity across Kernels

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

The underlying kernels exhibit a wide range of arithmetic intensities, from $\frac{1}{1000}$ths to 100s of floating-point operations per byte, *i.e.* 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 ML, data analytics and graphics [27, 32]. Convolution is a critical component in image processing [3] and convolutional neural networks [59]. Fast Fourier Transform (FFT) is widely used in speech and image processing for signal transformation [6, 79]. Sparse matrix-matrix multiplication (SpMM) is an important irregular kernel in graph analytics (part of GraphBLAS [54]), scientific computation [9, 24, 117], and problems involving big data with sparse connections [45, 93]. Another common sparse operation is sparse matrix-vector multiplication (SpMV), which is predominant in graph algorithms such as PageRank and Breadth-First Search [77], as well as ML-driven text analytics [5].

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 in a *single 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. Cache and scratchpad memory (SPM) are two well-known and extensively researched types of on-chip memory [8, 58, 110]. 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. In this case, a private configuration delivers better performance.

2.3.3 Dataflow: Demand-Driven vs. Spatial. In this work, we refer to demand-driven dataflow as the dataflow used by GPPs, wherein cores use on-demand loads/stores to read/write data and communicate via shared memory. In contrast, spatial dataflow architectures (*e.g.* systolic arrays) are data-parallel designs consisting of multiple processing elements (PEs) with direct PE-to-PE channels. Each PE receives data from its neighbor(s), performs an operation, and passes the result to its next neighbor(s) [60]. If pipelined correctly, this form of data orchestration harnesses the largest degree of parallelism. However, it is harder to map and write efficient software for certain applications on spatial architectures [49].

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. This motivates the intuition that

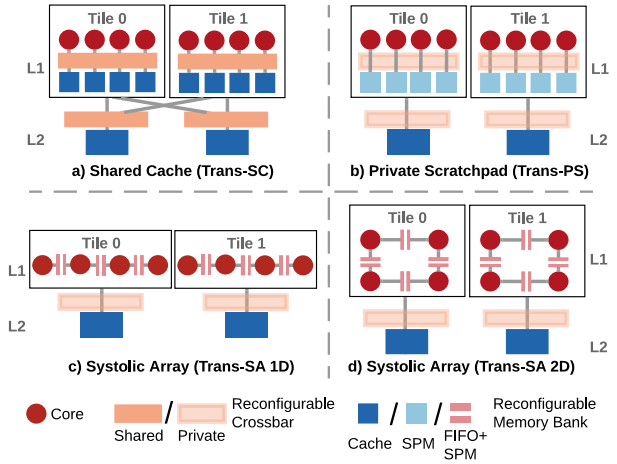

Figure 3: High-level Transmuter architecture showing the configurations evaluated in this work, namely a) Trans-SC (L1: shared cache, L2: shared cache), b) Trans-PS (L1: private SPM, L2: private cache), and c, d) Trans-SA (L1: systolic array, L2: private cache).

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

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

an architecture that reconfigures between these designs can accelerate diverse workloads that exhibit a spectrum of characteristics.

3 HIGH-LEVEL ARCHITECTURE

The takeaways from the previous section are the fundamental design principles behind our proposed architecture, `Transmuter`. `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 these tile-heterogeneous 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 the L1 memory banks and the tiles to the L2 banks, 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 L1 cache banks 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 the cores within each tile, and is suited for highly data parallel applications where the work is relatively balanced between the cores. `Transmuter` supports both 1D and 2D systolic configurations. Note that the L2 is configured as a cache for the same reason as with Trans-PS.

We omit an exhaustive evaluation of all possible `Transmuter` 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 [66]. 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 small processor with floating-point (FP) and load/store (LS) units that uses a standard ISA. Its small footprint enables `Transmuter` to incorporate many such GPEs within standard reticle sizes. The large number of GPEs coupled with miss status holding registers (MSHRs) in the cache hierarchy allows `Transmuter` to exploit memory-level parallelism (MLP) across the sea of cores. 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 is primarily responsible for distributing work across GPEs, using either *static* (*e.g.* greedy) or *dynamic* scheduling (*e.g.* skipping GPEs with full queues), thus trading-off code complexity for work-balance.

4.2 Work and Status Queues

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

4.3 Reconfigurable Data Cache

`Transmuter` has two layers of multi-banked memories, called reconfigurable data caches, *i.e.* R-DCaches (Fig. 4 – b, c). Each R-DCache bank is a standard cache module with enhancements to support the following modes of operation:

- **CACHE.** Each bank is accessed as a non-blocking, write-back, write-no-allocate cache with a least-recently used replacement

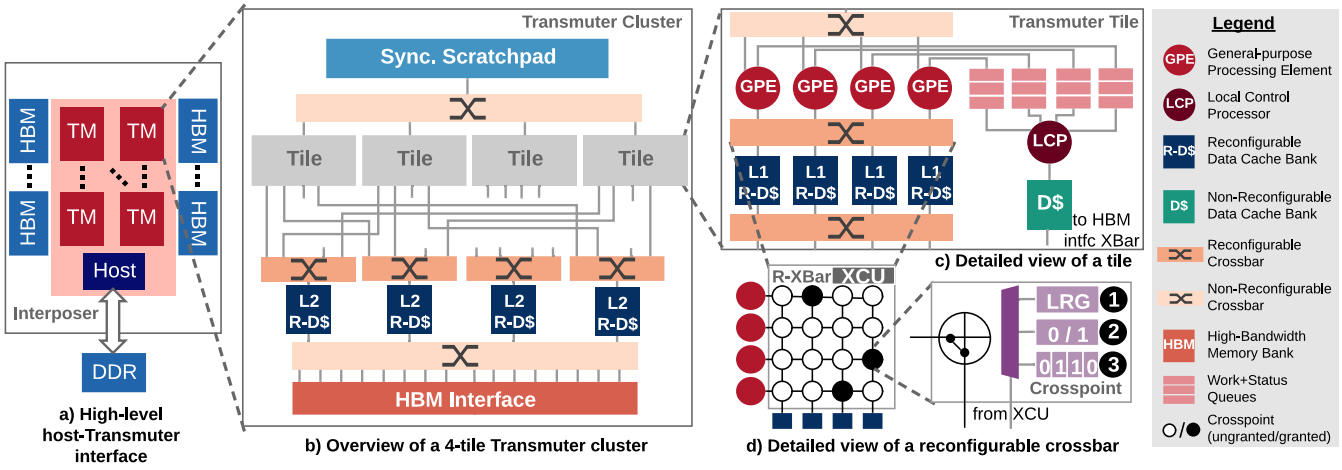

Figure 4: a) High-level overview of a host-Transmuter system. b) Transmuter architecture showing 4 tiles and 4 L2 R-DCache banks, along with L2 R-XBars, the synchronization SPM and interface to off-chip memory. Some L2 R-XBar input connections are omitted for clarity. c) View of a single tile, showing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

policy. The banks are interleaved at a set-granularity, and a cache-line physically resides in one bank. Additionally, this mode uses a simple stride prefetcher to boost performance for regular kernels.
- **SPM.** The tag array, set-index logic, prefetcher and MSHRs are powered off and the bank is accessed as a scratchpad.
- **FIFO+SPM.** A partition of the bank is configured as SPM, while the remainder are accessed as FIFO queues (Fig. 5 – left), using a set of head/tail pointers. The queue depth can be reconfigured using memory-mapped registers. The low-level abstractions for accessing the FIFOs are shown in Fig. 5 (right). This mode is used to implement spatial dataflow in Trans-SA (Fig. 3).

4.4 Reconfigurable Crossbar

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 [48, 99]. These and other work [2] 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 contended accesses to the same port get serialized. Arbitration is done in a single cycle using a least-recently granted policy [99], while the serialization latency varies between 0 and $(N_{src} - 1)$ cycles. This mode is used in Trans-SC.
- **TRANSPARENT.** A requester can only access its corresponding resource, *i.e.* the crosspoints within the crossbar are set to 0 or 1 (Fig. 4-d). Thus, the R-XBar is transparent and incurs *no* arbitration or serialization delay in this mode. Trans-PS (in L1 and L2) and Trans-SA (in L2) employ TRANSPARENT R-XBars.
- **ROTATE.** The R-XBar cycles through a set of one-to-one port connections programmed into the crosspoints. This mode also has no crossbar arbitration cost. Fig. 6 illustrates how port multiplexing is used to emulate spatial dataflow in a 1D systolic array configuration (Trans-SA).

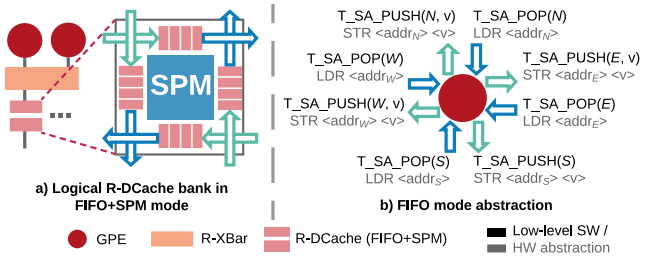

Figure 5: a) Logical view of an R-DCache bank in FIFO+SPM mode, showing 4 FIFO partitions, one for each direction in 2D. b) Loads and stores to special addresses corresponding to each direction are mapped to POP and PUSH calls, respectively, into the FIFOs.

There are two L1 R-XBars within a tile (Fig. 4-c). The upper R-XBar enables 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) in a program. With explicit synchronization, the programmer can instantiate a mutex in the synchronization SPM and protect the CS with it. The same can also be achieved through implicit synchronization, with the following sequence of events: ① both GPEs ← LCP, ② LCP → GPE`0`, ③ GPE`0` executes the CS, ④ GPE`0` → LCP, ⑤ LCP → GPE`1`, ⑥ GPE`1` executes the CS, ⑦ GPE`1` → LCP, where ← denotes `POP`-from and → is a `PUSH`-to the work or status queue.

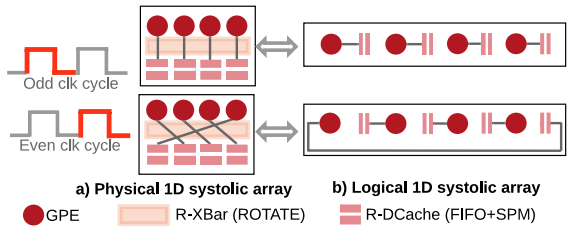

Figure 6: a) Physical and b) logical views of 1D systolic array connections within a Transmuter tile. Spatial dataflow is achieved by the R-XBar rotating between the two port-connection patterns.

Compared to traditional hardware coherence, these techniques reduce power through lower on-chip traffic [53, 87]. The synchronization SPM is interfaced to the LCPs and GPEs through a low-throughput two-level arbiter tree, as accesses to this SPM were not bottleneck for any of the evaluated workloads.

4.6 Miscellaneous Reconfiguration Support

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

4.7 Reconfiguration Overhead

`Transmuter` can self-reconfigure at run-time (initiated by an LCP) if the target configuration is known *a priori*. Reconfiguration can also be initiated by the host using a command packet with relevant metadata. The programming interface used to initiate this is discussed in Sec. 6. Each step of the hardware reconfiguration happens in parallel and is outlined below.

- **GPE.** Upon receiving the reconfiguration command, GPEs switch the base/bound registers that their LS units are connected to (Sec. 4.6) in a single cycle.
- **R-XBar.** ARBITRATE ↔ TRANSPARENT reconfiguration entails a 1-cycle latency, as it only switches MUXes in the R-XBar (Fig. 4-d). The ROTATE mode uses set/unset patterns, which requires a serial transfer of bit vectors from on-chip registers (*e.g.* a 64×64 design incurs a 6-cycle latency[2]).
- **R-DCache.** Switching from CACHE to SPM mode involves a 1-cycle toggle of the scratchpad controller. The FIFO+SPM mode involves programming the head and tail pointer for each logical FIFO queue, which are transferred from control registers (4 cycles for 4 FIFO partitions).

Thus, the net reconfiguration time, accounting for buffering delays, amounts to *~10 cycles*, which is faster than FPGAs and many CGRAs (Sec. 2.1). For host-initiated reconfiguration, overheads associated with host-to-`Transmuter` communication leads to a net reconfiguration time of few 10s of cycles. We limit our discussions to self-reconfiguration in this work. Since `Transmuter` does not implement hardware coherence, switching between certain `Transmuter` configurations entails cache flushes from L1 to L2, from L2 to HBM, or both. The levels that use the SPM or FIFO+SPM mode do not need flushing. Furthermore, our write-no-allocate caches circumvent flushing for streaming workloads that write output data only once. Even when cache flushes are inevitable, the overhead is small (<1% of execution time) for the evaluated kernels in Sec. 8.

[2]Latency (in cycles) = ceil($N_{\text{rotate\_patterns}} \times N_{\text{dst}} \times \log_2(N_{\text{src}})$ / xfer_width)

5 KERNEL MAPPING

`Transmuter` is built using COTS cores that lend the architecture to be kernel-agnostic. Here, we present our mappings of the fundamental kernels in Sec. 2 on the selected `Transmuter` configurations. Code snippets for three of our implementations are listed in Appendix C. Additional kernels in the domain of linear algebra have been mapped and evaluated on a preliminary version of `Transmuter` for different resource sharing configurations [101].

We 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 [69]. We similarly implement `GeMM` on Trans-PS but with the blocked partial results stored in the private L1 SPMs. Naturally, Trans-PS misses the opportunity for data sharing. For Trans-SA, the GPEs execute `GeMM` in a systolic fashion with the rows of A streamed through the L2 cache, and the columns of B loaded from the L1 SPM.

`GeMV`. `GeMV` is 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 avoid network deadlock in our `GeMM` and `GeMV` 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 $(2 \cdot F^2 \cdot N^2 \cdot IC \cdot OC)/S$ FLOPS, for an $F \times F$ filter convolving with stride S over an $N \times N$ image, with IC input and OC output channels. The filter is reused while computing one output channel, and across multiple images. Input reuse is limited to $O(F \cdot OC)$, for $S<F$. On Trans-SC, we assign each GPE to compute the output of multiple rows, to maximize the filter reuse across GPEs. For Trans-PS and Trans-SA, we statically partition each image into $B \times B \times IC$ sub-blocks, such that the input block and filter fit in the private L1 SPM. Each block is then mapped to a GPE for Trans-PS, and to a set of F adjacent GPEs of a 1D systolic array for Trans-SA using a row stationary approach similar to [17].

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 [25, 46]. Our Trans-SA mapping is similar to pipelined systolic ASICs; each stage is assigned to a single GPE, and each GPE immediately pushes its outputs to its neighbor. The butterflies in each stage are computed greedily. To reduce storage and increase parallelism, Trans-SA uses run-time twiddle coefficient generation when the transform size is too large for on-chip memory, *e.g.* >256 for 2×8, 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 decrease with increasing sparsity, *e.g.* $\sim 2N^3 r_M^2$, for uniform-random $N \times N$ matrices with density r_M. Furthermore, sparse storage formats lead to indirection and thus irregular memory accesses [65, 87]. We implement SpMM in Trans-SC using a prior outer product approach [87]. 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 operates similarly, but with the sorting list placed in private L1 SPM, given that SPMs are a better fit for operations on disjoint memory chunks. Lastly, SpMM in Trans-SA is implemented following a recent work that uses sparse packing [41]. Both the columns of A and rows of B are packed in memory. The computation is equally split across the tiles.

SpMV. SpMV, similar to SpMM, is bandwidth-bound and produces low FLOPS ($\sim 2N^2 r_M r_v$ for a uniformly random $N \times N$ matrix with density r_M, and vector with density r_v). We exploit the low memory traffic in the outer product algorithm for sparse vectors, mapping it to Trans-SC and Trans-PS. The GPEs and LCPs collaborate to merge the partial product columns in a tree fashion, with LCP `0` writing out the final elements to the HBM. SpMV on 1D Trans-SA is implemented using inner product on a packed sparse matrix as described in [41]. The packing algorithm packs 64 rows as a slice, and assigns one slice to each 1×4 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 within the sub-tile.

Finally, for both SpMM and SpMV, we use dynamic scheduling for work distribution to the GPEs (Sec. 4.1), in order to exploit the amenability of sparse workloads to SPMD architectures [87].

6 PROTOTYPE SOFTWARE STACK

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

The highest level API, called `TransPy`, is a drop-in replacement for the well-known high-performance Python library `NumPy`, *i.e.*

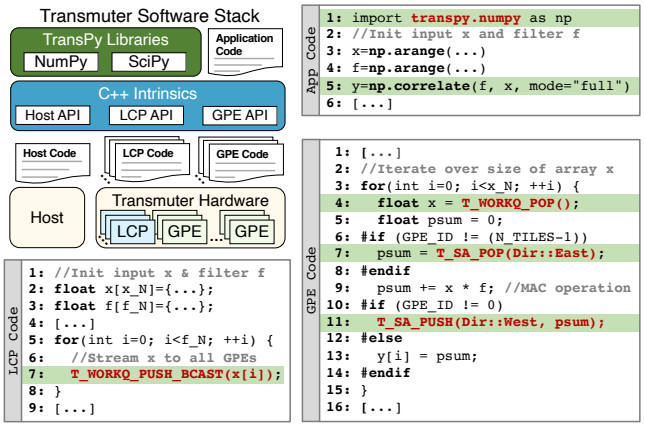

Figure 7: Transmuter software stack. Application code is written using Python and invokes library code for the host, LCPs and GPEs. The implementations are written by experts using our C++ intrinsics library. Also shown is an example of a correlation kernel on Trans-SA (host library code not shown). The end-user writes standard `NumPy` code and changes only the import package to `transpy.numpy` (App:L1). Upon encountering the library call (App:L5), the host performs data transfers and starts execution on Transmuter. The LCP broadcasts the vector `x` to all GPEs (LCP:L7). Each GPE pops the value (GPE:L4), performs a MAC using its filter value (`f`) and east neighbor's partial sum (GPE:L7), and sends its partial sum westward (GPE:L11). The last GPE stores the result into HBM. The host returns control to the application after copying back the result vector `y`.

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 precompiled kernel library is invoked. We use `pybind11` [47] as the abstraction layer between Python and `C++`. `TransPy` also contains drop-in replacements for `SciPy`, `PyTorch`, `NetworkX`, and other libraries used in scientific computing, ML, graph analytics, *etc.*

`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 [104], 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 [80]. These calls allow for synchronization at different granularities, such as globally, within tiles, and across 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 for expert programmers to write their own implementations, such as sophisticated heterogeneous implementations that partition the work between the host CPU and `Transmuter`. As an alternative to writing hand-tuned

Table 2: Microarchitectural parameters of Transmuter gem5 model.

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

Table 3: Specifications of baseline platforms and libraries evaluated.

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

kernels for `Transmuter`, we are actively working on prototyping a compiler to automatically generate optimized `C++`-level library code for `Transmuter` based on the Lift data-parallel language [103], the details of which are left for a future work.

7 EXPERIMENTAL METHODOLOGY

This section describes the methodology used to derive performance, power and area estimates for `Transmuter`. Tab. 2 shows the parameters used for modeling `Transmuter`. We compare `Transmuter` with a high-end Intel Core i7 CPU and NVIDIA Tesla V100 GPU running optimized commercial libraries. The baseline specifications and libraries are listed in Tab. 3. For fair comparisons, we evaluate two different `Transmuter` designs, namely **`TransX1`** and **`TransX8`**, that are each comparable in area to the CPU and GPU, respectively. `TransX1` has a single 64×64 `Transmuter` cluster and `TransX8` employs 8 such clusters. Both designs have one HBM2 stack/cluster to provide sufficient bandwidth and saturate all GPEs in the cluster.

7.1 Performance Models

We used the `gem5` simulator [10, 11] to model the `Transmuter` hardware. We modeled the timing for GPEs and LCPs after an in-order Arm Cortex-M4F, and cache and crossbar latencies based on a prior chip prototype that uses SSN crossbars [88, 90]. Data transfer/set-up times are excluded for all platforms. Throughput is reported in FLOPS/s and only accounts for useful (algorithmic) FLOPS.

The resource requirement for simulations using this detailed `gem5` model is only tractable for `Transmuter` systems up to 8×16. For larger systems, we substitute the `gem5` cores with trace replay engines while retaining the `gem5` model for the rest of the system. Offline traces are generated on a native machine and streamed through these engines. This allows us to simulate systems up to one 64×64 cluster. On average, across the evaluated kernels, the trace-driven model is pessimistic to 4.5% of the execution-driven model. For a multi-cluster system, we use analytical models from `gem5`-derived bandwidth and throughput scaling data (Sec. 8.2).

We implemented each kernel in `C++` and hand-optimized it for each `Transmuter` configuration using the intrinsics discussed in Sec. 6. Compilation was done using an Arm GNU compiler with the `-O2` flag. All experiments used single-precision FP arithmetic.

7.2 Power and Area Models

We designed RTL models for `Transmuter` hardware blocks and synthesized them. The GPEs and LCPs are modeled as Arm Cortex-M4F cores. For the R-XBar, we use the SSN design proposed in [99], augmented with an XCU. The R-DCaches are cache modules enhanced with SPM and FIFO control logic.

The crossbar and core power models are based on RTL synthesis reports and the Arm Cortex-M4F specification document. The R-XBar power model is calibrated against the data reported in [99]. For the caches and synchronization SPM, we used CACTI 7.0 [81] to estimate the dynamic energy and leakage power. We further verified our power estimate for `SpMM` on `Transmuter` against a prior `SpMM` ASIC prototype [88], 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 [86] and measured memory bandwidth, and subtracted it out. The power is scaled for iso-technology comparisons using quadratic scaling.

8 EVALUATION

We evaluate the Trans-SC, Trans-PS and Trans-SA configurations on the kernels in Sec. 5. We then compare the best-performing `Transmuter` to the CPU and GPU, and deep-dive into the evaluation of an application that exercises rapid reconfiguration. Lastly, we show comparisons with prior platforms and power/area analysis.

8.1 Performance with Different Configurations

Fig. 8 presents relative comparisons between Trans-SC, Trans-PS and Trans-SA in terms of performance. This analysis was done on a small 2×8 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 the work imbalance across GPEs.

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

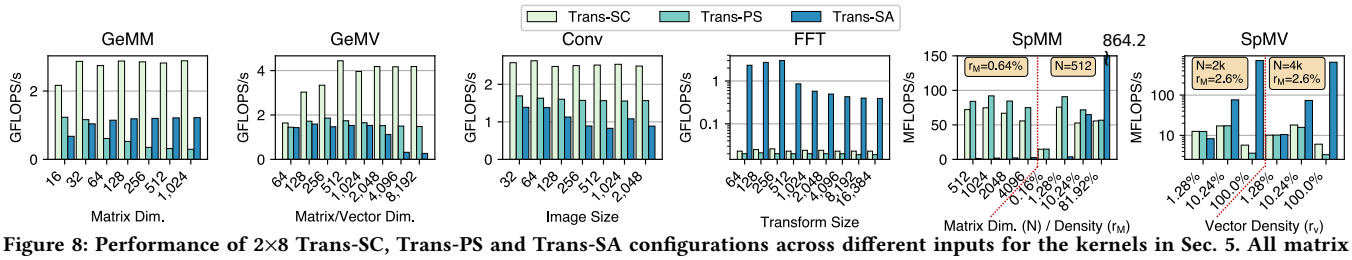

Figure 8: Performance of 2×8 Trans-SC, Trans-PS and Trans-SA configurations across different inputs for the kernels in Sec. 5. All matrix operations are performed on square matrices without loss of generality. Conv uses 3×3 filters, 2 input/output channels, and a batch size of 2.

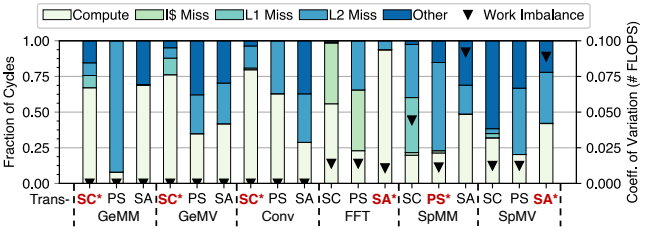

Figure 9: Cycle breakdown for the kernels in Sec. 5. * (red) indicates the best-performing configuration. "Other" comprises of stalls due to synchronization and bank conflicts. ▼: work imbalance across GPEs (σ/μ of # FLOPS). Inputs are: 1k (GeMM), 8k (GeMV), 2k (Conv), 16k (FFT), 4096, 0.64% (SpMM), 4k, 2.6%, dense vector (SpMV).

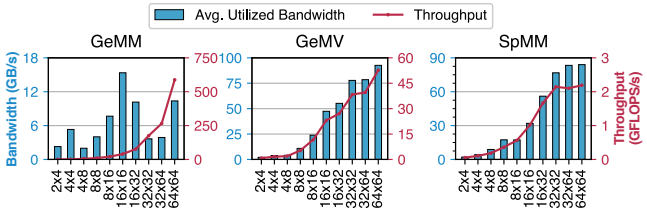

Figure 10: Effect of scaling tiles and GPEs per tile on performance and memory bandwidth for GeMM (Trans-SC), GeMV (Trans-SC) and SpMM (Trans-PS). Inputs are: 1k (GeMM), 8k (GeMV), 4096, 0.64% (SpMM).

For FFT, Trans-SA achieves significantly higher throughput because it benefits from the streaming inputs and exploits better data reuse, evidenced by ~10× less memory bandwidth usage compared to Trans-SC/Trans-PS. Inter-GPE synchronization and coherence handling at the end of each stage limit the performance for Trans-SC/Trans-PS. In addition, the control flow in the non-systolic code is branchy and contributes to expensive ICache misses. Trans-SA performs better for sizes <512 compared to other sizes, as the twiddle coefficients are loaded from on-chip rather than being computed.

For SpMM, the multiply phase of outer product is better suited to Trans-SC as the second input matrix rows are shared. The merge phase is amenable to Trans-PS since the private SPMs overcome the high thrashing that Trans-SC experiences while merging multiple disjoint lists. Trans-SA dominates for densities >~11%, however it performs poorly in comparison to outer product for highly-sparse matrices. Although ~50% of the time is spent on compute operations (Fig. 9), most of these are wasted on fetched data that are discarded after failed index matches. For SpMV, performance depends on the input matrix size, dimensions, as well as the vector density. Notably, Trans-SA benefits through the spatial dataflow for SpMV but not for SpMM, because the SpMV implementation treats the vector as dense, and thus can stream-in the vector elements efficiently into the GPE arrays. At sufficiently high vector sparsities, outer product on Trans-SC/Trans-PS outperforms Trans-SA by avoiding fetches of zero-elements. For higher densities, they suffer from the overhead of performing mergesort that involves frequent GPE-LCP synchronization, and serialization at LCP 0.

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

8.2 Throughput and Bandwidth Analysis

We investigate here the impact of scaling the number of tiles (N_T) and GPEs per tile (N_G) for an N_T×N_G Transmuter. Fig. 10 illustrates the scaling of Transmuter for GeMM, GeMV and SpMM. GeMM shows near-linear performance scaling with the GPE-count. The bandwidth utilization, however, does not follow the same trend as it is dependent on the data access pattern at the shared L2 R-DCache that influences the L2 hit-rate. GeMV exhibits increased bank conflicts in the L1 shared cache upon scaling up N_G, *e.g.* from 32×32 to 32×64. Thus, the performance scaling shows diminishing returns with increasing N_G, but scales well with increasing N_T. SpMM performance scales well until the bandwidth utilization is close to peak, at which point bank conflicts at the HBM controllers restrict further gains. SpMV follows the trend of GeMV, while FFT and Conv, show near-linear scaling with increasing system size (not shown).

We also discuss some takeaways from our cache bandwidth analysis for the best-performing Transmuter configuration. GeMM exhibits a high L1 utilization (20.4%) but low L2 utilization (2.7%), as most of the accesses are filtered by the L1. In contrast, SpMM and SpMV in Trans-PS and Trans-SA modes, respectively, have higher L2 utilizations of 68.5-90.5%. The linear algebra kernels show a relatively balanced utilization across the banks, with the coefficient of variation ranging from 0-10.1%. In contrast, both FFT and Conv have a skewed utilization, due to the layout of twiddle coefficients in the SPM banks for FFT, and the small filter size for Conv.

8.3 Design Space Exploration

We performed a design space exploration with the mapped kernels to select R-DCache sizes for Transmuter. Sizes of 4 kB per bank for both L1 and L2 show the best energy-efficiency for all kernels except SpMV. SpMV in Trans-SA benefits from a larger L2 private cache that lowers the number of evictions from fetching discrete packed matrix rows (recall that in Trans-SA, all GPEs in a tile access the same L2 bank). Other kernels achieve slim speedups with larger cache capacities. The dense kernels already exhibit good hit rates

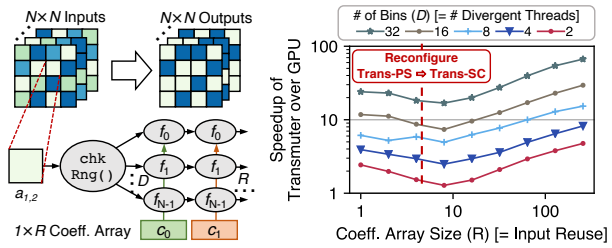

Figure 11: *Left:* A synthetic parallel application that launches threads to process $N \times N$ matrices. Each thread (*i*) reads the input value and bins it into one of D bins, (*ii*) applies R instances of function f_d unique to bin d and writes the result. Each element of a *coefficient* array feeds into f_d. Thus the input is reused R times and the degree of divergence scales with D. *Right:* Speedup of Transmuter with a uniform-random matrix (# GPEs = # GPU threads = 64). Transmuter reconfigures from Trans-PS to Trans-SC beyond $R = 4$.

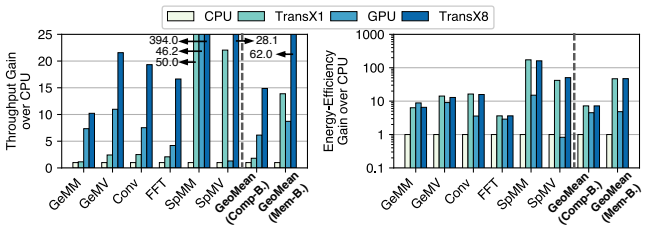

Figure 12: Throughput (left) and energy-efficiency (right) improvements of Transmuter over the CPU and GPU. Data is averaged across the inputs: 256-1k (GeMM), 2k-8k (GeMV), 512-2k (Conv), 4k-16k (FFT), 1k-4k, 0.64% (SpMM), and 2k-4k, 2.6% (r_M), 10.2%-100% (r_v) (SpMV). Geometric mean improvements for the compute-bound and memory-bound kernels are shown separately.

due to blocking and prefetching in Trans-SC. SpMM is bottlenecked by cold misses due to low reuse. FFT has a 3.0× speedup with 64 kB L1/L2, compared to 4 kB L1/L2, as the number of coefficients stored on-chip scales with L1 size. But, this is outweighed by a 6.4× increase in power. Other parameters such as work and status queue depths were chosen to be sufficiently large such that the GPEs are never idled waiting on the LCP.

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 around the architectural advantages of Transmuter over a GPU for applications with notable contrast in these characteristics. We implement a parallel microbenchmark on Transmuter and the GPU that allows independent tuning of the divergence and reuse. Fig. 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 this microbenchmark with a batch of 1,000 32×32 images on a 4×16 Transmuter design, and compare it with the GPU running 64 threads (2 warps, inputs in shared memory) to ensure fairness. Fig. 11 (right) presents two key observations:

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

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

8.5 Comparison with the CPU and GPU

We now compare the best-performing Transmuter configuration with the CPU and GPU running optimized commercial 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, as discussed in Sec. 7.

Compute-Bound Kernels (GeMM, Conv, FFT). TransX1 harnesses high data-level parallelism, and thus achieves performance improvements of 1.2-2.5× over the CPU, despite clocking at $\frac{1}{4}$th the speed of the deeply-pipelined CPU cores. The true benefit of Transmuter's simple cores and efficient crossbars appear in the form of energy-efficiency gains, ranging from 3.6-16.3×, which is owed largely to the high power consumption of the bulky out-of-order CPU cores. Over the GPU, TransX8 gets performance gains of 1.3-2.6× and efficiency improvements of 0.8-4.4× with an efficient implementation on Trans-SC for GeMM and Conv. The ~20% energy-efficiency loss for GeMM is explained by the amenability of GeMM 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 the spatial dataflow of Trans-SA, with each tile operating on an independent input stream, thus leading to minimum conflicts. The gap between throughput gain (4.0×) and energy-efficiency gain (1.3×) over the GPU is explained by the cuFFT algorithm that is more efficient for batched FFTs.

Memory-Bound Kernels (GeMV, SpMM, 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 1,024. The 14.2× energy-efficiency gain of TransX1 stems from tuning down the number of active GPEs to curtail bandwidth-starvation, thus saving power.

On SpMM and SpMV, the performance of Transmuter is highly sensitive to the densities and sizes of the inputs, with improvements ranging from 4.4-110.8× over the CPU and 5.9-37.7× over the GPU. With SpMM, execution in Trans-PS enables overcomes the CPU's limitation of an inflexible cache hierarchy, as well as harnesses high MLP across the sea of GPEs. While Transmuter is memory-bottlenecked for SpMM, SpMV is bounded by the scheduling granularity of 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

Table 4: Estimated speedups for the end-to-end workloads in Fig. 2.

Speedup	DANMF	LSTM	Marian	MaxCut	MFCC
TransX1 *vs.* CPU	4.1×	1.1×	2.2×	6.2×	1.7×
TransX8 *vs.* GPU	3.5×	3.8×	2.1×	7.2×	1.6×

	NBSGD	RolePred	SemSeg	Sinkhorn	VidSeg
TransX1 *vs.* CPU	3.5×	2.7×	2.4×	3.1×	2.2×
TransX8 *vs.* GPU	2.8×	2.3×	2.5×	3.0×	2.8×

achieves just 0.6% and 0.002% of its peak performance, respectively for SpMM and SpMV, impaired by memory and synchronization stalls. In comparison, SPMD on Transmuter reduces synchronization, resulting in 21-42% time spent on useful computation (Fig. 9). For SpMM, the outer product implementation demonstrates ASIC-level performance gains of 5.9-11.6× [87] over the GPU, by minimizing off-chip traffic and exploiting the asynchronicity between GPEs. As with GeMV, disabling bandwidth-starved resources contributes to the energy-efficiency gains.

Effect of Iso-CPU Bandwidth. TransX1 uses one HBM stack that provides 125 GB/s peak bandwidth, about 3.6× greater than the DDR3 bandwidth to the CPU. If given the bandwidth of the DDR3 memory, TransX1 still achieves performance gains averaging 17.4× and 6.3× for SpMM and SpMV, respectively. For GeMV, TransX1 remains within a modest 6-8% 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 (*e.g.* CPU→GPU) data transfer times are excluded for all platforms. Overall, 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 [63, 97]. Sinkhorn computation typically involves large, sparse matrices in conjunction with dense matrices. We implement the algorithm described in [18]; see Appendix B. The inner loop has two major kernels: a GeMM operation masked by a sparse weight matrix (M-GeMM), and a dense matrix - sparse matrix multiplication (DMSpM).

The mapping on Transmuter is shown in Fig. 13. M-GeMM uses a variation of blocked-GeMM, 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-GeMM and DMSpM-Multiply exhibit the best performance in Trans-SC configuration, due to good data reuse across GPEs. In contrast, DMSpM-Merge has optimal performance on Trans-PS, exhibiting a 84.9-98.3% speedup (not shown in figure) 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-GeMM 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-GeMM but not DMSpM, as DMSpM uses

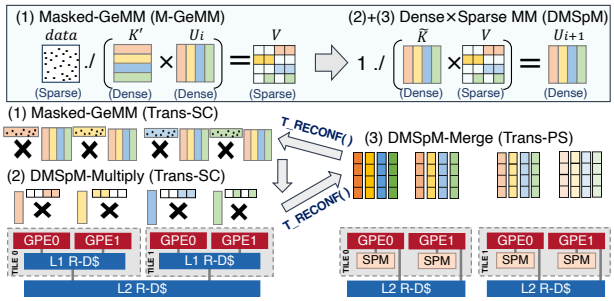

Figure 13: Mapping of a multi-kernel, mixed data application, Sinkhorn, on Transmuter. Computation iterates between M-GeMM and DMSpM, with Trans-SC ↔ Trans-PS reconfiguration before and after DMSpM-Merge. DMSpM-Merge benefits from the private SPMs in Trans-PS, since each GPE works on multiple disjoint lists.

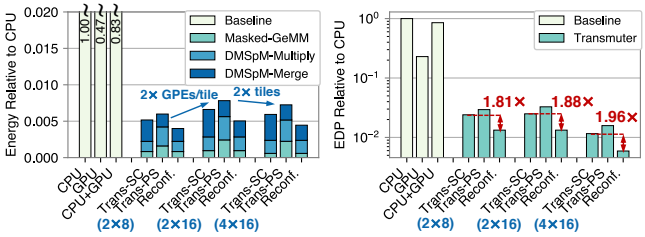

Figure 14: Per inner-loop iteration energy (left) and EDP (right) comparing Trans-SC, Trans-PS and Reconf. (Trans-SC ↔ Trans-PS) for Sinkhorn normalized to CPU. Input matrix dimensions and densities are — *query*: (8k×1), 1%, *data*: (8k×1k), 1%, *M*: (8k×8k), 99%.

a streaming algorithm. Overall, dynamic reconfiguration results in 47.2% and 96.1% better performance and energy-delay product (EDP), respectively, over Trans-SC-only for the 4×16 Transmuter. A heterogeneous solution is also compared against, where M-GeMM is done on the CPU and DMSpM on the GPU, but this implementation is bottlenecked by CPU → GPU data transfers. As derived from Figure 14, the 4×16 Transmuter achieves 38.8× and 144.4× lower EDP than the GPU and heterogeneous solutions, respectively.

8.7 Comparison with Other Platforms

Tab. 5 shows the estimated energy-efficiency improvements of Transmuter over recent FPGA, CGRA and ASIC implementations. The efficiencies reported in prior work are scaled quadratically for iso-technology comparisons with Transmuter. Overall, Transmuter achieves average efficiency gains of 3.4× 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 1.7% of the CPU's area. The estimated worst-case reconfiguration overhead is 74.9 nJ.

Table 5: Energy-efficiency improvements (black) and deteriorations (red) of Transmuter over prior FPGAs, CGRAs and ASICs.

Platform	GeMM	GeMV	Conv	FFT	SpMM	SpMV
FPGA	2.7× [34]	8.1× [64][3]	2.7× [119]	2.2× [34]	3.6× [35]	3.0× [23]
CGRA	2.2× [95]	3.0× [19]	1.2× [19]	1.0× [52][4]	1.9× [19]	2.9× [19]
ASIC	(32.1×) [91]	(10.5×) [98]	(13.8×) [109] (7.6×) [98]	(18.1×) [92] (17.0×) [26]	(3.0×) [88] (4.1×) [120]	(3.9×) [87]

[3]Performance/bandwidth used as power is N/A. [4]Estimated for floating-point based on [108].

Table 6: Power and area of a 64×64 Transmuter cluster in 14 nm.

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

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 different dataflows, reconfiguring faster at a coarser granularity, and supporting a COTS ISA/compiler.

Reconfigurability. A few prior work reconfigure at the sub-core level [19, 44, 55, 76, 95] and the network-level [37, 56, 83, 107]. In contrast, `Transmuter` uses native in-order cores and the reconfigurablity lies in the memory and interconnect. Some recent work propose reconfiguration at a coarser granularity [4, 19, 70, 95]. 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 [107] implements a tiled architecture focusing on developing an efficient, distributed interconnect. Stream Dataflow [83] and SPU [19] reconfigure at runtime, albeit with non-trivial overheads to initialize the Data-Flow Graph (DFG) configuration. `Transmuter`, on the other hand, relies on flexible memories and interconnect that enable fast on-the-fly reconfiguration, thus catering to the nature of the application.

Flexibility. Prior work has also delved into efficient execution across a wide range of applications. Plasticine [95] is a reconfigurable accelerator for parallel patterns, consisting of a network of Pattern Compute/Memory Units (custom SIMD FUs/single-level SPM) that can be reconfigured at compile-time. Stream Dataflow [83] is a new computing model that efficiently executes algorithms expressible as DFGs, with inputs/outputs specified as streams. The design comprises a control core with stream scheduler and engines, interfaced around a custom, pipelined FU-based CGRA. SPU [19] targets data-dependence using a stream dataflow model on a reconfigurable fabric composed of decomposable switches and PEs that split networks into finer sub-networks. The flexibility of `Transmuter` stems from the use of general-purpose cores and the reconfigurable memory subsystem that morphs the dataflow and on-chip memory, thus catering to both inter- and intra-workload diversity.

Table 7: Qualitative comparison with prior work [19, 39, 83, 95, 107].

Architecture	PE Compute Paradigm	Dataflow	Compiler Support	Reconfig. Granularity	On-chip Memory
Plasticine	SIMD	Spatial	DSL	Pipeline-level, compile-time	SPM
Stream Dataflow	SIMD	Stream	ISA extn.	Network-level, run-time	SPM+FIFO
SPU	SIMD	Stream	ISA extn.	Network-/ Sub-PE-level, run-time	Compute-enabled SPM+FIFO
Ambric	MIMD/ SPMD	Demand-driven	Custom	Network-level, run-time	SPM+FIFO
RAW	MIMD/ SPMD	Demand-driven	Modified COTS	Network-level, run-time	Cache
`Transmuter` [this work]	**MIMD/ SPMD**	**Demand-driven/ Spatial**	**COTS**	**Network-/ On-chip-memory-level, run-time**	**Reconfig. Cache/SPM/ SPM+FIFO**

Programmability. There have been proposals for programmable CGRAs that abstract the low-level hardware. Some work develop custom programming models, such as Rigel [53] and MaPU [112]. Others extend an existing ISA to support their architecture, such as Stitch [105] and LACore [102]. Plasticine [95] uses a custom DSL called 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. This fabric supports fast reconfiguration of the memory type, resource sharing and dataflow, thus tailoring `Transmuter` to the nature of the workload. We also presented a software stack comprised of drop-in replacements for standard Python libraries. We demonstrated `Transmuter`'s performance and efficiency on a suite of fundamental kernels, as well as mixed data-based multi-kernel applications. Our evaluation showed average energy-efficiency improvements of 46.8× (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

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[2] Nilmini Abeyratne, Reetuparna Das, Qingkun Li, Korey Sewell, Bharan Giridhar, Ronald G. Dreslinski, David Blaauw, and Trevor Mudge. 2013. Scaling Towards Kilo-core Processors with Asymmetric High-Radix Topologies. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 496–507.

[3] Edward H Adelson, Charles H Anderson, James R Bergen, Peter J Burt, and Joan M Ogden. 1984. Pyramid Methods in Image Processing. *RCA Engineer* 29, 6 (1984), 33–41.

[4] Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, Massoud Pedram, and Muhammad Shafique. 2019. X-CGRA: An energy-efficient approximate coarse-grained reconfigurable architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019).

[5] Anima Anandkumar, Rong Ge, Daniel J. Hsu, Sham M. Kakade, and Matus Telgarsky. 2012. Tensor decompositions for learning latent variable models. *CoRR* abs/1210.7559 (2012). arXiv:1210.7559

[6] Tuba Ayhan, Wim Dehaene, and Marian Verhelst. 2014. A 128~2048/1536 point FFT hardware implementation with output pruning. In *2014 22nd European Signal Processing Conference (EUSIPCO)*. IEEE, 266–270.

[7] David F Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA programming for the masses. *Commun. ACM* 56, 4 (2013), 56–63.

[8] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. 2002. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002*. IEEE, 73–78.

[9] Nathan Bell, Steven Dalton, and Luke N Olson. 2012. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152.

[10] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7.

[11] Nathan L Binkert, Ronald G Dreslinski, Lisa R Hsu, Kevin T Lim, Ali G Saidi, and Steven K Reinhardt. 2006. The M5 simulator: Modeling networked systems. *Ieee micro* 26, 4 (2006), 52–60.

[12] Ian Buck. 2010. The evolution of GPUs for general purpose computing. In *Proceedings of the GPU Technology Conference 2010*. 11.

[13] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 141–151.

[14] Sergi Caelles, Kevis-Kokitsi Maninis, Jordi Pont-Tuset, Laura Leal-Taixé, Daniel Cremers, and Luc Van Gool. 2017. One-Shot Video Object Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 221–230.

[15] Benton Highsmith Calhoun, Joseph F Ryan, Sudhanshu Khanna, Mateja Putic, and John Lach. 2010. Flexible circuits and architectures for ultralow power. *Proc. IEEE* 98, 2 (2010), 267–282.

[16] Web Chang. 2001. Embedded configurable logic ASIC. US Patent 6,260,087.

[17] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2016), 127–138.

[18] Marco Cuturi. 2013. Sinkhorn Distances: Lightspeed Computation of Optimal Transport. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'13)*. 2292–2300.

[19] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. ACM, 924–939.

[20] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rakesh K Gupta, Zhiru Zhang, Ronald G Dreslinski, Christopher Batten, and Michael B Taylor. 2018. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro* 38, 2 (2018), 30–41.

[21] Chris HQ Ding, Xiaofeng He, Hongyuan Zha, Ming Gu, and Horst D Simon. 2001. A min-max cut algorithm for graph partitioning and data clustering. In *Proceedings 2001 IEEE International Conference on Data Mining*. IEEE, 107–114.

[22] Claire Donnat, Marinka Zitnik, David Hallac, and Jure Leskovec. 2018. Learning structural node embeddings via diffusion wavelets. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 1320–1329.

[23] Richard Dorrance, Fengbo Ren, and Dejan Marković. 2014. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. ACM, 161–170.

[24] Iain S Duff, Michael A Heroux, and Roldan Pozo. 2002. 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 (2002), 239–267.

[25] E Swartzlander Earl Jr. 2006. Systolic FFT Processors: Past, Present and Future. In *IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*. IEEE, 153–158.

[26] Nasim Farahini, Shuo Li, Muhammad Adeel Tajammul, Muhammad Ali Shami, Guo Chen, Ahmed Hemani, and Wei Ye. 2013. 39.9 GOPS/Watt Multi-Mode CGRA Accelerator for a Multi-Standard Basestation. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*. IEEE, 1448–1451.

[27] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. 2004. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 133–137.

[28] Siying Feng, Subhankar Pal, Yichen Yang, and Ronald G Dreslinski. 2019. Parallelism Analysis of Prominent Desktop Applications: An 18-Year Perspective. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 202–211.

[29] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. 2015. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing* 71, 10 (2015), 3934–3957.

[30] Florian Fricke, André Werner, Keyvan Shahin, and Michael Hübner. 2018. CGRA tool flow for fast run-time reconfiguration. In *International Symposium on Applied Reconfigurable Computing*. Springer, 661–672.

[31] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. 2013. Data transfer matters for GPU computing. In *2013 International Conference on Parallel and Distributed Systems*. IEEE, 275–282.

[32] Noriyuki Fujimoto. 2008. Dense matrix-vector multiplication on the CUDA architecture. *Parallel Processing Letters* 18, 04 (2008), 511–530.

[33] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Ieee, 126–137.

[34] Heiner Giefers, Raphael Polig, and Christoph Hagleitner. 2016. Measuring and Modeling the Power Consumption of Energy-Efficient FPGA Coprocessors for GEMM and FFT. *Journal of Signal Processing Systems* 85, 3 (Dec. 2016), 307–323.

[35] Heiner Giefers, Peter Staar, Costas Bekas, and Christoph Hagleitner. 2016. 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)*. IEEE, 46–56.

[36] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matthew Moe, and R Reed Taylor. 2000. PipeRench: A reconfigurable architecture and compiler. *Computer* 33, 4 (2000), 70–77.

[37] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically specialized datapaths for energy efficient computing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 503–514.

[38] Azzam Haidar, Mark Gates, Stan Tomov, and Jack Dongarra. 2013. Toward a scalable multi-gpu eigensolver via compute-intensive kernels and efficient communication. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 223–232.

[39] Tom R Halfhill. 2006. Ambric's new parallel processor. *Microprocessor Report* 20, 10 (2006), 19–26.

[40] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 620–629.

[41] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. 2020. Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices. In *Proceedings of the 34th ACM International Conference on Supercomputing (ICS '20)*. ACM, 12.

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

[43] Randall A Hughes and John D Shott. 1986. The future of automation for high-volume wafer fabrication and ASIC manufacturing. *Proc. IEEE* 74, 12 (1986), 1775–1793.

[44] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. 2007. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '07)*. ACM, 186–197.

[45] Satoshi Itoh, Pablo Ordejón, and Richard M Martin. 1995. Order-N tight-binding molecular dynamics on parallel computers. *Computer physics communications* 88, 2-3 (1995), 173–185.

[46] Preston A. Jackson, Cy P. Chan, Jonathan E. Scalera, Charles M. Rader, and M. Michael Vai. 2004. A systolic FFT architecture for real time FPGA systems," High Performance Embedded Computing Conference (HPEC04. In *In High Performance Embedded Computing Conference (HPEC04.*

[47] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. 2017. pybind11–Seamless operability between C++ 11 and Python. https://github.com/pybind/pybind11

[48] Supreet Jeloka, Reetuparna Das, Ronald G Dreslinski, Trevor Mudge, and David Blaauw. 2014. Hi-Rise: a high-radix switch for 3D integration with single-cycle arbitration. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 471–483.

[49] Kurtis T. Johnson, Ali R Hurson, and Behrooz Shirazi. 1993. General-purpose systolic arrays. *Computer* 26, 11 (1993), 20–31.

[50] 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. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, 1–12.

[51] Marcin Junczys-Dowmunt, Roman Grundkiewicz, Tomasz Dwojak, Hieu Hoang, Kenneth Heafield, Tom Neckermann, Frank Seide, Ulrich Germann, Alham Fikri Aji, Nikolay Bogoychev, André F. T. Martins, and Alexandra Birch. 2018. Marian: Fast neural machine translation in C++. *arXiv preprint arXiv:1804.00344* (2018).

[52] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[53] John Kelm, Daniel Johnson, Matthew Johnson, Neal Crago, William Tuohy, Aqeel Mahesri, Steven Lumetta, Matthew Frank, and Sanjay Patel. 2009. Rigel: An Architecture and Scalable Programming Interface for a 1000-Core Accelerator. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 140–151.

[54] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy G. Mattson, and José E. Moreira. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*. IEEE, 1–9.

[55] Khubaib, M. Aater Suleman, Milad Hashemi, Chris Wilkerson, and Yale N. Patt. 2012. MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP. *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (2012), 305–316.

[56] Martha Mercaldi Kim, John D. Davis, Mark Oskin, and Todd Austin. 2008. Polymorphic On-Chip Networks. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, 101–112.

[57] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, 296–311.

[58] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. 2015. Stash: Have Your Scratchpad and Cache It Too. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. ACM, 707–719.

[59] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[60] Hsiang-Tsung Kung. 1982. Why systolic architectures? *IEEE computer* 15, 1 (1982), 37–46.

[61] Ian Kuon and Jonathan Rose. 2007. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on computer-aided design of integrated circuits and systems* 26, 2 (2007), 203–215.

[62] Ian Kuon, Russell Tessier, and Jonathan Rose. 2008. *FPGA architecture: Survey and challenges*. Now Publishers Inc.

[63] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger. 2015. From Word Embeddings to Document Distances. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37* (Lille, France) *(ICML'15)*. 957–966.

[64] Georgi Kuzmanov and Mottaqiallah Taouil. 2009. Reconfigurable sparse/dense matrix-vector multiplier. In *2009 International Conference on Field-Programmable Technology*. IEEE, 483–488.

[65] Benjamin C Lee, Richard W Vuduc, James W Demmel, and Katherine A Yelick. 2004. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *International Conference on Parallel Processing, 2004. ICPP 2004.* IEEE, 169–176.

[66] Chang-Chi Lee, CP Hung, Calvin Cheung, Ping-Feng Yang, Chin-Li Kao, Dao-Long Chen, Meng-Kai Shih, Chien-Lin Chang Chien, Yu-Hsiang Hsiao, Li-Chieh Chen, Michael Su, Michael Alfano, Joe Siegel, Julius Din, and Bryan Black. 2016. An overview of the development of a GPU with integrated HBM on silicon interposer. In *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*. IEEE, 1439–1444.

[67] Chang-Hwan Lee. 2015. A gradient approach for value weighted classification learning in naive Bayes. *Knowledge-Based Systems* 85 (2015), 71–79.

[68] Dongwook Lee, Manhwee Jo, Kyuseung Han, and Kiyoung Choi. 2009. FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability. In *2009 International Conference on Field-Programmable Technology*. IEEE, 376–379.

[69] Jiajia Li, Xingjian Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2012. An optimized large-scale hybrid DGEMM design for CPUs and ATI GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 377–386.

[70] Cao Liang and Xinming Huang. 2008. SmartCell: A power-efficient reconfigurable architecture for data streaming applications. In *2008 IEEE Workshop on Signal Processing Systems*. IEEE, 257–262.

[71] Leibo Liu, Dong Wang, Min Zhu, Yansheng Wang, Shouyi Yin, Peng Cao, Jun Yang, and Shaojun Wei. 2015. An energy-efficient coarse-grained reconfigurable processing unit for multiple-standard video decoding. *IEEE Transactions on Multimedia* 17, 10 (2015), 1706–1720.

[72] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. 2019. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. *ACM Computing Surveys (CSUR)* 52, 6 (2019), 1–39.

[73] Beth Logan. 2000. Mel Frequency Cepstral Coefficients for Music Modeling. In *ISMIR*, Vol. 270. 1–11.

[74] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully Convolutional Networks for Semantic Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 3431–3440.

[75] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. 2016. Asic clouds: Specializing the datacenter. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 178–190.

[76] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. 2000. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (Vancouver, British Columbia, Canada) *(ISCA '00)*. ACM, New York, NY, USA, 161–171.

[77] Tim Mattson, David A. Bader, Jonathan W. Berry, Aydin Buluç, Jack J. Dongarra, Christos Faloutsos, John Feo, John R. Gilbert, Joseph Gonzalez, Bruce Hendrickson, Jeremy Kepner, Charles E. Leiserson, Andrew Lumsdaine, David A. Padua, Stephen Poole, Steven P. Reinhardt, Mike Stonebraker, Steve Wallach, and Andrew Yoo. 2013. Standards for graph algorithm primitives. In *IEEE High Performance Extreme Computing Conference, HPEC 2013, Waltham, MA, USA, September 10-12, 2013*. IEEE, 1–2.

[78] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2018. An analysis of neural language modeling at multiple scales. *arXiv preprint arXiv:1803.08240* (2018).

[79] Badri Narayan Mohapatra and Rashmita Kumari Mohapatra. 2017. FFT and sparse FFT techniques and applications. In *2017 Fourteenth International Conference on Wireless and Optical Communications Networks (WOCN)*. IEEE, 1–5.

[80] Frank Mueller. 1993. Pthreads library interface. *Florida State University* (1993).

[81] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* 27 (2009), 28.

[82] Chris Nicol. 2017. A coarse grain reconfigurable array (cgra) for statically scheduled data flow computing. *Wave Computing White Paper* (2017).

[83] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, 416–429.

[84] Molly A O'Neil and Martin Burtscher. 2014. Microarchitectural performance characterization of irregular GPU kernels. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 130–139.

[85] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. 2015. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper* 2, 11 (2015), 1–4.

[86] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W Keckler, and William J Dally. 2017. Fine-grained DRAM: energy-efficient DRAM for extreme bandwidth systems. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 41–54.

[87] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.

[88] Subhankar Pal, Dong-Hyeon Park, Siying Feng, Paul Gao, Jielun Tan, Austin Rovinski, Shaolin Xie, Chun Zhao, Aporva Amarnath, Timothy Wesley, Jonathan Beaumont, Kuan-Yu Chen, Chaitali Chakrabarti, Michael Bedford Taylor, Trevor N. Mudge, David T. Blaauw, Hun-Seok Kim, and Ronald G. Dreslinski. 2019. A 7.3 M Output Non-Zeros/J Sparse Matrix-Matrix Multiplication Accelerator using Memory Reconfiguration in 40 nm. In *2019 Symposium on VLSI Circuits, Kyoto, Japan, June 9-14, 2019*. IEEE, 150.

[89] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. 2019. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 304–315.

[90] Dong-Hyeon Park, Subhankar Pal, Siying Feng, Paul Gao, Jielun Tan, Austin Rovinski, Shaolin Xie, Chun Zhao, Aporva Amarnath, Timothy Wesley, Jonathan Beaumont, Kuan-Yu Chen, Chaitali Chakrabarti, Michael Bedford Taylor, Trevor N. Mudge, David T. Blaauw, Hun-Seok Kim, and Ronald G. Dreslinski. 2020. A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix-Matrix Multiplication Accelerator. *Journal of Solid-State Circuits* 55, 4 (2020), 933–944.

[91] Ardavan Pedram, Andreas Gerstlauer, and Robert A Van De Geijn. 2011. A high-performance, low-power linear algebra core. In *ASAP 2011-22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE, 35–42.

[92] Ardavan Pedram, John D. McCalpin, and Andreas Gerstlauer. 2014. A Highly Efficient Multicore Floating-Point FFT Architecture Based on Hybrid Linear Algebra/FFT Cores. *Journal of Signal Processing Systems* 77, 1 (01 Oct 2014), 169–190.

[93] Gerald Penn. 2006. Efficient transitive closure of sparse matrices over closed semirings. *Theoretical Computer Science* 354, 1 (2006), 72–81.

[94] Kara KW Poon, Steven JE Wilton, and Andy Yan. 2005. A detailed power model for field-programmable gate arrays. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 10, 2 (2005), 279–302.

[95] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 389–402.

[96] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 13–24.

[97] Tim Salimans, Han Zhang, Alec Radford, and Dimitris Metaxas. 2018. Improving GANs using optimal transport. *arXiv preprint arXiv:1803.05573* (2018).

[98] Fabian Schuiki, Michael Schaffner, and Luca Benini. 2019. Ntx: An energy-efficient streaming accelerator for floating-point generalized reduction workloads in 22 nm fd-soi. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 662–667.

[99] Korey Sewell, Ronald G. Dreslinski, Thomas Manville, Sudhir Satpathy, Nathaniel Ross Pinckney, Geoffrey Blake, Michael Cieslak, Reetuparna Das, Thomas F. Wenisch, Dennis Sylvester, David T. Blaauw, and Trevor N. Mudge. 2012. Swizzle-Switch Networks for Many-Core Systems. *IEEE J. Emerg. Sel. Topics Circuits Syst.* 2, 2 (2012), 278–294.

[100] Muhammad Shafique and Siddharth Garg. 2016. Computing in the Dark Silicon Era: Current Trends and Research Challenges. *IEEE Design & Test* 34, 2 (2016), 8–23.

[101] Anuraag Soorishetty, Jian Zhou, Subhankar Pal, David Blaauw, H Kim, Trevor Mudge, Ronald Dreslinski, and Chaitali Chakrabarti. 2020. Accelerating Linear Algebra Kernels on a Massively Parallel Reconfigurable Architecture. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 1558–1562.

[102] Samuel Steffl and Sherief Reda. 2017. LACore: A Supercomputing-Like Linear Algebra Accelerator for SoC-Based Designs. In *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 137–144.

[103] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a Functional Data-Parallel IR for High-Performance GPU Code Generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 74–85.

[104] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 3 (2010), 66–73.

[105] Cheng Tan, Manupa Karunaratne, Tulika Mitra, and Li-Shiuan Peh. 2018. Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 575–587.

[106] Masakazu Tanomoto, Shinya Takamaeda-Yamazaki, Jun Yao, and Yasuhiko Nakashima. 2015. A cgra-based approach for accelerating convolutional neural networks. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE, 73–80.

[107] Michael Bedford Taylor, Jason Sungtae Kim, Jason E. Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul R. Johnson, Jae W. Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matthew I. Frank, Saman P. Amarasinghe, and Anant Agarwal. 2002. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro* 22, 2 (2002), 25–35.

[108] Vaishali Tehre, Pankaj Agrawal, and RV Kshrisagar. [n.d.]. Implementation of Fast Fourier Transform Accelerator on Coarse Grain Reconfigurable Architecture. ([n. d.]).

[109] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. 2017. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 13–26.

[110] Manish Verma, Lars Wehmeyer, Peter Marwedel, and Peter Marwedel. 2004. Cache-Aware Scratchpad Allocation Algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2 (DATE '04)*. IEEE Computer Society, 21264–.

[111] Kizheppatt Vipin and Suhaib A Fahmy. 2018. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–39.

[112] 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. 2016. MaPU: A novel mathematical computing architecture. *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016), 457–468.

[113] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. 2015. Enabling FPGAs in Hyperscale Data Centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. IEEE, 1078–1086.

[114] Mark Wijtvliet, Luc Waeijen, and Henk Corporaal. 2016. Coarse Grained Reconfigurable Architectures in the Past 25 years: Overview and Classification. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. IEEE, 235–244.

[115] Xilinx [n.d.]. *Partial Reconfiguration User Guide UG702 (v13.3)*. Xilinx. https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/ug702.pdf

[116] Xilinx [n.d.]. *Partial Reconfiguration User Guide UG909 (v2018.1)*. Xilinx. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug909-vivado-partial-reconfiguration.pdf

[117] Ichitaro Yamazaki and Xiaoye S Li. 2010. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *International Conference on High Performance Computing for Computational Science*. Springer, 421–434.

[118] Fanghua Ye, Chuan Chen, and Zibin Zheng. 2018. Deep Autoencoder-like Nonnegative Matrix Factorization for Community Detection. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 1393–1402.

[119] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 161–170.

[120] Qiuling Zhu, Tobias Graf, H. Ekin Sumbul, Lawrence T. Pileggi, and Franz Franchetti. 2013. Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware. *2013 IEEE High Performance Extreme Computing Conference (HPEC)* (2013), 1–6.

A LOW-LEVEL PROGRAMMING INTERFACE

Table 8: Critical host- and Transmuter-side C++ intrinsics used to write optimized kernel libraries (TID = Tile ID, GID = GPE ID). Note that the API is depicted for a single-cluster design, for simplicity.

Host-side Intrinsic Signature	Description
H_INIT()	Initialize host-Transmuter interface
H_LAUNCH()	Trigger Transmuter to start executing the kernel
H_FINISH()	Wait (block) until Transmuter finishes executing
H_SEND_DATA(&dst,&src,size)	Mem-copy from external DRAM to HBM
H_RETR_DATA(&dst,&src,size)	Mem-copy from HBM to external DRAM
H_SET_†_ARG(argID,&arg,TID,[GID])	Copy an argument to an LCP or GPE
H_COMPILE_BIN(path_to_bin,flags)	Dynamically compile GPE/LCP code
H_LD_BIN_†(&bin,TID,[GID])	Stream compiled GPE/LCP binary into the HBM
H_SYNC_ALL()	Synchronize with all LCPs and GPEs
H_RECONF(en_flag,TID,[GID])	Dynamically enable/disable GPE/LCP
H_RECONF<level>(config)	Trigger R-DCache/XBar reconfiguration
H_CLEANUP()	Teardown host interface and deallocate structures

Transmuter-side Intrinsic Signature	Description
T_LD_WORD(addr)	Read a word from SPM; addr determines the bank
T_ST_WORD(addr,val)	Write a word into SPM; addr determines the bank
T_SA_POP(direction)	Pop data from systolic neighbor GPE
T_SA_PUSH(direction,val)	Push data to systolic neighbor GPE
T_+Q_PUSH(val,[GID])	Push data to work/status queue
T_+Q_POP([GID])	Pop data from work/status queue
T_FREE_WORKQ_PUSH(val)	Push to the work queue of a free GPE
T_WORKQ_PUSH_BCAST(val)	Broadcast to all work queues in the tile
T_FLUSH<level>(bank)	Flush dirty data from to the next level
T_SPM_BOT<level,config>()	Get a pointer to bottom of R-DCache/Sync. SPM
T_SPM_TOP<level,config>()	Get a pointer to top of R-DCache/Sync. SPM
T_SYNC_LCPS()	Synchronize with all LCPs in Transmuter
T_SYNC_TILE()	Synchronize with all GPEs and LCP in the tile
T_SYNC_ALL()	Synchronize with all LCPs, GPEs and host
T_SLEEP()	Put self into sleep to conserve power
T_RECONF<level>(self_flag,config)	Self-reconfigure R-DCache/XBar / wait for host

†: LCP/GPE +: WORK/STATUS

B SINKHORN ALGORITHM

Algorithm B.1 Sinkhorn Distance (MATLAB syntax)

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

C SELECTED KERNEL IMPLEMENTATIONS

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

function GEMV_LCP($start$, end, N_G)
▷ $start$: start row index, end: end row index, N_G: num. GPEs per tile
  $gid = 0$;
  for $row \leftarrow start$ to end **do**
    T_WORKQ_PUSH(gid, row);
    $gid = (gid == N_G - 1)\ ?\ 0 : (gid + 1)$;
  end for
  T_WORKQ_PUSH_BCAST(-1);
end function
function GeMV _GPE($A, B, C, N, \alpha, \beta$) ▷ $C = \alpha \cdot A * B + \beta \cdot C$, N: matrix dim.
  while (row = T_WORKQ_POP()) != -1 **do**
    $psum = 0$;
    for $col \leftarrow 0$ to $N - 1$ **do**
      $psum += A[row][col] * B[col]$;
    end for
    $C[row] = \beta * C[row] + \alpha * psum$;
  end while
end function

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

function SPMV_LCP()
  T_WORKQ_PUSH_BCAST(1);
end function
function SPMV_GPE($A_{rowID}, A_{colID}, A_{val}, A_{part}, B, B_{part}, C, N, P$)
▷ $C = A * B$, N: matrix/vector dim., N_G: num. GPEs per tile, N_T: num. tiles
  T_WORKQ_POP();
  $parts_{per\_tile} = \text{ceil}(N/N_T)$; ▷ partitions per tile
  $i = A_{part}[gid * N_T * N_G + tid]$
  for $part \leftarrow tid * parts_{per\_tile}$ to $(tid + 1) * parts_{per\_tile}$ **do**
    $b_{start} = B_{part}[part][gid]$; ▷ tid: tile ID, gid: GPE ID
    $b_{end} = B_{part}[part][gid + 1]$;
    $sp = sp_{start}$ = T_SPM_BOT<Lev::L1, Conf::systolic_array_1d>();
    for $j \leftarrow b_{start}$ to b_{end} **do**
      T_ST_WORD(sp++, $B[j]$);
    end for
    $sp_{sum} = sp$;
    for $row \leftarrow part * P$ to $(part + 1) * P$ **do** ▷ P: row partition per tile
      $psum = 0$;
      while $A_{rowID}[i] == row$ **do**
        b = T_LD_WORD($sp_{start} + A_{colID}[i]$);
        $psum += A_{val}[i$++$] * b$;
      end while
      T_ST_WORD(sp++, $psum$);
    end for
    for $row \leftarrow 0$ to P **do**
      $popped = (gid\ !=\ 0)\ ?$ T_SA_POP(Dir::West) $: 0$;
      $sum = popped$ + T_LD_WORD($sp_{sum} + row$);
      if $gid == N_G - 1$ **then**
        $C[part * P + row] = sum$;
      else
        T_SA_PUSH(Dir::East, sum);
      end if
    end for
  end for
end function

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

function FFT_LCP($input$, $output$, N, is_input, is_output)
▷ N: FFT size, $\log_2(N)$: num. FFT stages
  if is_input **then**
    for $i \leftarrow 0$ to $N - 1$ **do**
      T_WORKQ_PUSH(0, $input[i]$);
    end for
  end if
  if is_output **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 FFT_GPE($input$, $output$, N_G, N, S, P)
▷ N_G: num. GPEs per tile, S: step size, P: next step size
  $id = gid + tid * N_G$;
  sp = T_SPM_BOT<Lev::L1, Conf::systolic_array_1d>();
  for $i \leftarrow 0$ to $N/2$ **do**
    $in_1 = (id == 0)\ ?$ T_WORKQ_POP() $:$ T_SA_POP(Dir::West);
    $in_2 = (id == 0)\ ?$ T_WORKQ_POP() $:$ T_SA_POP(Dir::West);
    out_1, out_2 = compute_butterfly(in_1, in_2);
    if $id == \log_2(N) - 1$ **then**
      T_STATUSQ_PUSH(out_1); T_STATUSQ_PUSH(out_2);
    else
      T_ST_WORD($sp + i, out_1$); T_ST_WORD($sp + i + S, out_2$);
      if $i > P - 1$ **then**
        T_SA_PUSH(Dir::East, T_LD_WORD($sp + i - P$));
        T_SA_PUSH(Dir::East, out_1);
      end if
    end if
  end for
  for $i \leftarrow 0$ to P **do**
    T_SA_PUSH(Dir::East, T_LD_WORD($sp + S + i$));
    T_SA_PUSH(Dir::East, T_LD_WORD($sp + S + P + i$));
  end for
end function