Transmuter: Bridging the Efficiency Gap using Memory and Dataflow Reconfiguration

Citation for published version:

Digital Object Identifier (DOI):
10.1145/3410463.3414627

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
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.

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

ACM Reference Format:

1 INTRODUCTION
The past decade has seen a surge in emerging applications that are composed of multiple kernels4 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.

This work refers to kernels as the building blocks of larger applications.
Reconfigurability. Enabling near-ASIC efficiencies by morphing the hardware to specific kernel characteristics, for applications that are composed of multiple cascaded cores.

Programmability. Facilitating better adoption of non-GPP hardware by providing high-level software abstractions that are familiar to end-users and domain experts, and that mask the details of the underlying reconfigurable hardware.

To this end, we propose Transmuter, a reconfigurable accelerator that adapts to the nature of the kernel through a flexible fabric of lightweight cores, and reconfigurable memory and interconnect. Worker cores are grouped into tiles that are orchestrated by a control core. All cores support a standard ISA, thus allowing the hardware to be fully kernel-agnostic. Transmuter overcomes inefficiencies in vector processors such as GPUs for irregular applications [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 interwoven 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
workloads are derived from an ongoing multi-university program that involves big data with sparse connections [45, 93]. Another common problem involves signal transformation [6, 79]. Sparse matrix-matrix multiplication (SpMM) is an important irregular kernel in machine learning [18, 19, 34, 67, 73, 75, 118] and is frequently used 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 10x over privatization. This is owing 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 GPUs, 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
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 \( N_T \times N_W \) Transmuter to describe a system with \( N_T \) tiles and \( N_W \) 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 I-Caches 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 pushes 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-a, 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 strategy.
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 serializied. 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).

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 → GPE0, (3) GPE0 executes the CS, (4) GPE0 → LCP, (5) LCP → GPE1, (6) GPE1 executes the CS, (7) GPE1 → LCP, where → denotes POP from and ← is a PUSH-to the work or status queue.
The GPE LS unit is augmented with logic to route packets to Transmuter
which can self-reconfigure at run-time (initiated by an LCP)

2 configurations entails cache flushes from L1 to L2, from L2 to HBM,
ment hardware coherence, switching between certain
Transmuter self-reconfiguration in this work. Since
reconfiguration time of few 10s of cycles. We limit our discussions to
sociated with host-to-CGRAs (Sec. 2.1). For host-initiated reconfiguration, overheads as-
lays, amounts to ~10 cycles

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 dis-
cussed 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 en-
tails a 1-cycle latency, as it only switches MUXes in the R-XBar (Fig. 4-d). The ROUTE mode uses set/unset patterns, which re-
quires a serial transfer of bit vectors from on-chip registers (e.g. a 64x64 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 as-
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 circum-
vent flushing for streaming workloads that write output data only once. Even when cache flushes are inevitable, the overhead is small (<1% of execution time) for the evaluated kernels in Sec. 8.

5 KERNEL MAPPING
Transmuter is built using COTS cores that lend the architecture to be kernel-agnostic. Here, we present our mappings of the fundamental kernels in Sec. 2 on the selected Transmuter configurations. Code snippets for three of our implementations are listed in Appen-
dix 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 im-
plementations 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 im-
plementations by reconfiguring the FIFO depth of the L1 R-DCache (Sec. 4.3) to allow for sufficient buffering.

Conv. Conv in 2D produces \((2 \cdot F^2 \cdot N^2 \cdot I \cdot O)/S\) FLOPS, for an
\(F \times F\) filter convolving with stride \(S\) over an \(N \times N\) image, with \(I\) input and \(O\) output channels. The filter is reused while computing one output channel, and across multiple images. Input reuse is limited to \(O(F \cdot O)\), for \(S \leq F\). 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 \(B \times B \times I \times C\) 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 oper-
ations. 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 2x8, with the trade-off of making the problem compute-bound. On Trans-SC, the butterfly operations are distributed evenly among GPEs to compute a stage in parallel. LCPs assign inputs and collect outputs from GPEs. All cores synchronize after each stage. For Trans-PS, the same scheduling is used and partial results are stored in the L1 SPM.

5.3 Sparse Matrix Multiplication

\textbf{SpMM}. SpMM is a memory-bound kernel with low FLOPS that decrease with increasing sparsity, e.g., $\sim 2N^3r_M^2$ for uniform-random $N \times N$ matrices with density $r_M$. Furthermore, sparse storage formats lead to indirection and thus irregular memory accesses [65, 87]. We implement SpMM in Trans-SC using a prior outer product work distribution to the GPEs (Sec. 4.1), in order to exploit the amenability of sparse workloads to SPMD architectures [87].

SpMV. SpMV, similar to SpMM, is bandwidth-bound and produces low FLOPS ($\sim 2N^2r_Mr_v$ for a uniformly random $N \times N$ matrix with density $r_M$, and vector with density $r_v$). We exploit the low memory traffic in the outer product algorithm for sparse vectors, mapping it to Trans-SC and Trans-PS. The GPEs and LCPs collaborate to merge the partial product columns in a tree fashion, with LCP $\Theta$ 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 1x4 sub-tile within a tile. Each GPE loads the input vector elements into SPM, fetches the matrix element and performs MAC operations, with the partial results being streamed to its neighbor 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. the TransPy API exactly mirrors that of NumPy. In the code example in Fig. 7, note that only one change is needed to convert the NumPy program to TransPy. The np.correlate function is trapped in TransPy, dispatched to the Transmuter host layer, and a pre-compiled kernel library is invoked. We use pybind11 [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

![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 TransSA (host library code not shown). The end-user writes standard NumPy code and changes only the import package to from 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.](image-url)
Table 2: Microarchitectural parameters of Transmuter gem5 model.

<table>
<thead>
<tr>
<th>Module</th>
<th>Microarchitectural Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPE/LCP</td>
<td>1-issue, 4-stage, in-order (Minotaur/GPU) 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)</td>
</tr>
<tr>
<td>Work/Status Queue</td>
<td>4 B, 4-entry FIFO buffer between each GPE and LCP within a tile, blocks loads if empty and stores if full</td>
</tr>
<tr>
<td>R-DCache (per bank)</td>
<td>4 kB, 4-way set-associative, 1-ported, non-coherent cache with 8 MSHRs and 64 block size, stride prefetcher of degree 2, word-granular (L1) / cache-line-granular (L2)</td>
</tr>
<tr>
<td>R-Xbar</td>
<td>4×Nsrc×Ndst non-coherent crossbar with 1-cycle response</td>
</tr>
<tr>
<td>SPM</td>
<td>FIFO+SPM: 4 kB, 1-ported, physically-addressed, 32-bit head and tail pointer registers</td>
</tr>
<tr>
<td>Sync. SPM</td>
<td>4 kB, 1-ported, physically-addressed scratchpad</td>
</tr>
<tr>
<td>Main Memory</td>
<td>1 HBMx2 stack: 16 64-bit pseudo-channel, each 4x8000 MB/s, 80-150 ns average access latency</td>
</tr>
</tbody>
</table>

Table 3: Specifications of baseline platforms and libraries evaluated.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Specifications</th>
<th>Library Name and Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5-2690, 4 core/threads at 4.0-4.4 GHz, 16 GB DDR3 memory (@ 34.1 GB/s, AVX, SSE4.2, 122 nm² (14 nm)</td>
<td>MEL 2013.3.222 (Gemm/GeLU/SpMM/SpGeLU), DNN 1.1.0 (Conv), FFTW 3.3 (FFT)</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Tesla V100, 32H CUDA cores at 1.25 GHz, 16 GB HBM2 memory at 900 GB/s, 815 nm² (12 nm)</td>
<td>cuBLAS v10 (GEMM/SGEMM), cuDNN v7.5.3 (Conv), cuFFT v10.0 (FFT), cuSPARSE v5.1 (SpMM), cuPARSEC v3.0 (SpGeLU)</td>
</tr>
</tbody>
</table>

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 2x8 system to stress the hardware. The results show that the best performing Transmuter configuration is kernel-dependent, and in certain cases also input-dependent. Fig. 9 shows the cycle breakdowns and the work imbalance across GPEs.

For GeMM, Trans-SC achieves high L1 hit rates (>99%), as efficient blocking leads to good data reuse. Trans-SC 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 GeMW, 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/GeMW, 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.
Demand-driven dataflow with shared caching outperforms Trans-SC as the second input matrix rows are shared. The merge forms other configurations for GeMM (Trans-SC), GeMV (Trans-SC) and SpMM (Trans-PS). Inputs are: 1k (GeMM), 8k (GeMV), 2k (Conv), 16k (FFT), 4096, 0.64% (SpMM), 4k, 2.6%, dense vector (SpMV).

For FFT, Trans-SA achieves significantly higher throughput because it benefits from the streaming inputs and exploits better data reuse, evidenced by ~10x 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 <312 compared to other sizes, as the twiddle coefficients are loaded from on-chip rather than being computed.

For SpMV, the multiply phase of outer product is better suited to Trans-SA because the private SPMs overcome the bandwidth and bank conflicts in the L1 shared cache upon scaling up NCG. GeMV shows the best energy-efficiency for all kernels.

8.2 Throughput and Bandwidth Analysis

We investigate here the impact of scaling the number of tiles (NT) and GPEs per tile (NG) for an NT×NG Transmuter. The scaling of Transmuter for GeMM, GeMV and SpMM 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 NC, e.g. from 32×32 to 32×64. Thus, the performance scaling shows diminishing returns with increasing NC, but scales well with increasing NT. SpMM performance scales well until the bandwidth utilization is close to peak, at which point bank conflicts at the HBM controllers restrict further gains. GeMV 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: 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 \( f_\beta \) unique to bin \( \beta \) and writes the result. Each element of a coefficient array feeds into \( f_\beta \). 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 \( R = 4 \) due to blocking and prefetching in Trans-SC. SpMV is bottlenecked by cold misses due to low reuse. FFT has a 3.0x 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 6.4x 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 32x32 images on a 4x16 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 7.4x 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 (GeMV, Conv, FFT). TransX1 harnesses high data-level parallelism, and thus achieves performance improvements of 1.2-2.5x over the CPU, despite clocking at \( 34 \) 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 3.6-16.3x, 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 1.3-2.6x and efficiency improvements of 0.8-4.4x with an efficient implementation on Trans-SC for GeMV and Conv. The ~20% energy-efficiency loss for GeMV is explained by the amenability of GeMV to a SIMT paradigm; although the performance is similar between SIMT and SPMD, SPMD incurs slightly larger energy costs associated with higher control overhead over SIMT. On FFT, Transmuter sustains consistent performance scaling using 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 (4.0x) and energy-efficiency gain (1.3x) over the GPU is explained by the cuFFT algorithm that is more efficient for batched FFTs.

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

On SpMV and SpMM, the performance of Transmuter is highly sensitive to the densities and sizes of the inputs, with improvements ranging from 4.4-110.8x over the CPU and 5.9-37.7x 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 SpMV, SpMV is bounded by the scheduling granularity of packing algorithm deployed on Trans-SA. Despite that, for SpMV, TransX1 outperforms both the CPU as well as the GPU that has 7.2x greater available bandwidth. In case of the GPU, while there are sufficient threads to saturate the SMs, the thread divergence in the SIMT model is the bottleneck. The GPU
achieves 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 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.

<table>
<thead>
<tr>
<th>Platform</th>
<th>GRPM Power (mW)</th>
<th>GRPM Area (mm²)</th>
<th>Conv Power (mW)</th>
<th>Conv Area (mm²)</th>
<th>FFT Power (mW)</th>
<th>FFT Area (mm²)</th>
<th>SPIR Power (mW)</th>
<th>SPIR Area (mm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>2741</td>
<td>0</td>
<td>2741</td>
<td>0</td>
<td>2741</td>
<td>0</td>
<td>2741</td>
<td>0</td>
</tr>
<tr>
<td>CGRA</td>
<td>3.0% [34]</td>
<td>3.0% [34]</td>
<td>0.1% [36]</td>
<td>0.1% [36]</td>
<td>1.8% [37]</td>
<td>1.8% [37]</td>
<td>1.8% [37]</td>
<td>1.8% [37]</td>
</tr>
<tr>
<td>ASIC</td>
<td>52.2% [38]</td>
<td>52.2% [38]</td>
<td>19.3% [39]</td>
<td>19.3% [39]</td>
<td>30.6% [40]</td>
<td>30.6% [40]</td>
<td>30.6% [40]</td>
<td>30.6% [40]</td>
</tr>
</tbody>
</table>

Table 6: Power and area of a 64x64 Transmuter cluster in 14 nm.

<table>
<thead>
<tr>
<th>Module</th>
<th>Power (mW)</th>
<th>Area (mm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPE Cores</td>
<td>361.3</td>
<td>238.5</td>
</tr>
<tr>
<td>LCP Cores</td>
<td>5.6</td>
<td>22.3</td>
</tr>
<tr>
<td>Sync. SPM</td>
<td>0.6</td>
<td>0.1</td>
</tr>
<tr>
<td>All R-Caches</td>
<td>2566.0</td>
<td>373.0</td>
</tr>
<tr>
<td>LCP DL-Caches</td>
<td>39.3</td>
<td>9.9</td>
</tr>
<tr>
<td>L1 R-D-Caches</td>
<td>2527.1</td>
<td>204.0</td>
</tr>
<tr>
<td>L2 R-D-Caches</td>
<td>37.4</td>
<td>18.3</td>
</tr>
<tr>
<td>L1 R-XBars</td>
<td>1575.8</td>
<td>2149.3</td>
</tr>
<tr>
<td>L2 R-XBars</td>
<td>36.9</td>
<td>14.8</td>
</tr>
<tr>
<td>MUXes/Arbiters</td>
<td>581.9</td>
<td>87.6</td>
</tr>
<tr>
<td>Memory Ctrlrs</td>
<td>47.5</td>
<td>129.0</td>
</tr>
<tr>
<td>Total</td>
<td>8.0 W</td>
<td>5.4 W</td>
</tr>
</tbody>
</table>

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 reconfigurability 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.

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


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.

<table>
<thead>
<tr>
<th>Host-side Intrinsics Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_INIT()</td>
<td>Initialize host-Transmuter interface</td>
</tr>
<tr>
<td>T_FINISH()</td>
<td>Wait (block) until Transmitter finishes executing</td>
</tr>
<tr>
<td>T_SEND_DATA(addr, src_size, src)</td>
<td>Mem-cop from external DRAM to HBM</td>
</tr>
<tr>
<td>T_RECV_DATA(addr, src_size, src)</td>
<td>Mem-cop from HBM to external DRAM</td>
</tr>
<tr>
<td>T_WAIT(argID, arg, TID, GID)</td>
<td>Copy an argument to an LCP or GPE</td>
</tr>
<tr>
<td>T_GET_PIN(path_to_PIN, flags)</td>
<td>Dynamically compute GPE/TC code</td>
</tr>
<tr>
<td>T_UDP_PIN(Pin, TID, GID)</td>
<td>Stream compiled GPE/TC binary into the HBM</td>
</tr>
<tr>
<td>T_SYNC_ALL()</td>
<td>Synchronize with all LCPs and GPEs</td>
</tr>
<tr>
<td>T_UPGRADE()</td>
<td>Update Transmuter IP to enable Trans-SA configuration</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transmitter-side Intrinsics Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_WORKQ_PUSH_BCAST(val)</td>
<td>Trigger R-DCache/Xbar reconfiguration</td>
</tr>
<tr>
<td>T_STATUSQ_PUSH(val)</td>
<td>Lookup host interface and transmute structures</td>
</tr>
<tr>
<td>T_STATUSQ_POP()</td>
<td>Shut down host interface and transmute structures</td>
</tr>
<tr>
<td>T_SPKI()</td>
<td>Synchronize with all LCPs and GPEs</td>
</tr>
<tr>
<td>T_SPKI_ALL()</td>
<td>Synchronize with all LCPs, GPEs and host</td>
</tr>
<tr>
<td>T_SLEEP()</td>
<td>Put self into deep sleep power</td>
</tr>
<tr>
<td>T_TLM_INIT()</td>
<td>Read a word from SPM, addr determines the bank</td>
</tr>
<tr>
<td>T_TLM_RECV(val, addr)</td>
<td>Write a word into SPM, addr determines the bank</td>
</tr>
<tr>
<td>T_PPI PUSH(direction, val)</td>
<td>Pop data from systolic neighbor GPE</td>
</tr>
<tr>
<td>T_PPI_RECV(val, (addr))</td>
<td>Pop data to systolic neighbor GPE</td>
</tr>
<tr>
<td>T_PPI PUSH(direction, val)</td>
<td>Push data to systolic neighbor GPE</td>
</tr>
<tr>
<td>T_PPI_RECV(val, (addr))</td>
<td>Push data from systolic neighbor GPE</td>
</tr>
<tr>
<td>T_PPI PUSH_BCAST(val)</td>
<td>Broadcast to all work queues in the tile</td>
</tr>
<tr>
<td>T_PPI_RECV_BCAST(val)</td>
<td>Broadcast to all work queues in the tile</td>
</tr>
<tr>
<td>T_PPI PUSH_TOLEVEL(config)</td>
<td>Get a pointer to top of R-DCache/Xbar SPM</td>
</tr>
<tr>
<td>T_PPI_RECV_LEVEL(config)</td>
<td>Get a pointer to R-DCache/Xbar SPM</td>
</tr>
<tr>
<td>T_PPI_SYNC_LCP(config)</td>
<td>Synchronize with all LCPs in Transmuter</td>
</tr>
<tr>
<td>T_PPI_SYNC_STATUSQ(config)</td>
<td>Synchronize with all GPEs and LCP in the tile</td>
</tr>
</tbody>
</table>

B SINKHORN ALGORITHM

Algorithm B.1 Sinkhorn Distance (MATLAB syntax)

function Sinkhorn(query, data, M, y, ε)
    o = size(M, 2);  \* M. distance matrix, y: regularization parameter, ε: tolerance
    H = ones(length(query, y) / length(query, y));  \* Masked-GeMM
    K = exp(-M / y);  \* DMSpM
    err = oo;  \* DMSpM
    while err > ε do
        V = data ./ (K'U);  \* Masked-GeMM
        U = 1 ./ (K V);  \* DMSpM
        err = sum(U * (U - prevU)) / sum(U)^2;
        end while
        return sinkhorn_distance(query, data, M, y, ε);
    end function

C SELECTED KERNEL IMPLEMENTATIONS

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

function GeMV_LCP(A, B, C, N, α, β)  \* C = α * A + β * C, N: matrix dim.
    \* start: start row index, end: end row index, N: num. GPEs per tile
    \* geMv: 0
    for row = 1 to N do
        \* end for
        \* T_WORKQ_PUSH_BCAST(-1);
        \* end function
        \* GeMV_GPE(data, M, y, ε)  \* GeMV_LCP(data, M, y, ε);
    \* end function

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

function SpMV_LCP(E, A, N, α, β, C, N, P)  \* C = A * B, N: matrix/vector dim., NC: num. GPEs per tile, N2: num. tiles
    \* partitions per tile
    \* i = A[i] \* \* N2 \* NC + tile \* end function
for row = 0 to N - 1 do
    \* end for
    \* T_WORKQ_PUSH_BCAST(-1);
    \* 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, log2(N): num. FFT stages
    \* is_input: true;
    \* end function
    \* if is_input then
        \* for i = 0 to N - 1 do
        \* T_WORKQ_PUSH_BCAST(0);
        \* end for
        \* if is_output then
        \* for i = 0 to N - 1 do
        \* T_WORKQ_PUSH_BCAST(1);
        \* end for
        \* end if
        \* end if
        \* end function

function FFT_GPE(input, output, N, S, P)
    \* NC: num. GPEs per tile, S: step size, P: next step size
    \* id = gid + tile + NC;
    \* sp = T_SPKI_BOT<level, config>Lev::L1, Conf::systolic_array_1d();
    \* end if
    \* end function