AFRL-RY-WP-TR-2023-0262

TRANSMUTER – A RECONFIGURABLE COMPUTER

D. Blaauw, R. Dreslinski, H.S. Kim, and T. Mudge
University of Michigan

M. Cole and M. O'Boyle
University of Edinburgh

C. Chakrabarti
Arizona State University

MAY 2024
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with The Under Secretary of Defense memorandum dated 24 May 2010 and AFRL/DSO policy clarification email dated 13 January 2020. This report is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2023-0262 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//Signature//

ASHLEY DEMANGE BRANDEWIE
Program Manager
Sensors Subsystems Branch
Aerospace Components & Subsystems Division

//Signature//

TIMOTHY R. JOHNSON, Chief
Sensors Subsystems Branch
AFRL Division Name
Aerospace Components & Subsystems Division

//Signature//

JOHN S. CETNAR (acting)
Deputy Chief
Aerospace Components & Subsystems Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
MAY 2024	Final	6 August 2018	31 August 2023

4. TITLE AND SUBTITLE

TRANSMUTER – A RECONFIGURABLE COMPUTER

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
FA8650-18-2-7864	N/A	N/A
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**
N/A	N/A	Y1UM

6. AUTHOR(S)

D. Blaauw, R. Dreslinski, H.S. Kim, and T. Mudge (University of Michigan)
M. Cole and M. O'Boyle (University of Edinburgh)
C. Chakrabarti (Arizona State University)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)			8. PERFORMING ORGANIZATION REPORT NUMBER
University of Michigan 500 S. State Street, Ann Arbor, MI 48109	University of Edinburgh Old College, South Bridge, Edinburgh EH8 9YL, UK	Arizona State University 350 E Lemon St, Tempe, AZ 85287	

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'SREPORT NUMBER(S)
Air Force Research Laboratory Sensors Directorate Wright-Patterson Air Force Base, OH 45433-7320 Air Force Materiel Command United States Air Forces	Defense Advanced Research Projects Agency (DARPA/MTO) 675 North Randolph Street Arlington, VA 22203	AFRL/RYDR	AFRL-RY-WP-TR-2023-0262

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

This material is based on research sponsored by the Air Force Research Laboratory (AFRL) and the 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 the Air Force Research Laboratory (AFRL), the Defense Advanced Research Projects Agency (DARPA), or the U.S. Government. Report contains color.

14. ABSTRACT

This report presents the final state of the DARPA Software Defined Hardware (SDH) program named Transmuter. The goal of SDH is to build a runtime-reconfigurable hardware and software that enables near ASIC performance without sacrificing programmability. Transmuter is a fast reconfigurable design consisting of a sea of tiny, in-order cores connected through a two-level cache-crossbar hierarchy to high-bandwidth off-chip memory. Transmuter v1.0 was fabricated in 28 nm CMOS and occupies 12 mm^2. Across kernels, median energy-efficiency improvements of 11.6× and 37.2× are achieved versus the CPU and GPU, respectively. The RTL design of the system has been validated on an FPGA prototype and is ready for transition to several potential future programs. The software met the programmability goals of the program and is available in several open-source software releases.

15. SUBJECT TERMS

Reconfigurable processor, cache-crossbar, general-purpose processing elements, reconfigurable interconnect fabric, dual-function memories, intelligent memory controller, CMOS, energy efficient hardware

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
Unclassified	Unclassified	Unclassified	SAR	137

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Ashley DeMange Brandewie	N/A

Table of Contents

Section **Page**

Section **Page**

Section **Page**

List of Figures

Figure **Page**

Figure **Page**

Figure **Page**

Figure **Page**

List of Tables

Table **Page**

1 EXECUTIVE SUMMARY

This report presents the final state of the DARPA SDH program named Transmuter, with PI's from the University of Michigan, University of Edinburgh, and Arizona State University. ARM was also a participating member of the program in the earlier phases, until the company underwent restructuring. The RTL design of our system has been validated on an FPGA prototype and is ready for transition to several potential future programs. The software met the programmability goals of the program and is available in several open-source software releases.

2 ARCHITECTURE OVERVIEW

Transmuter, at a high level, is a fast reconfigurable design consisting of a sea of tiny, in-order cores connected through a two-level cache-crossbar hierarchy to high-bandwidth off-chip memory. A high-level block diagram of the proposed architecture is shown below.

2.1 Detailed Architectural Description

The four basic building blocks of the proposed architecture are – General-purpose Processing Elements (GPEs), a reconfigurable interconnect fabric, dual-function memories, and an intelligent memory controller. A high-level diagram of the Transmuter architecture is presented in Figure 1. The following sub-sections present the design and functionality of each building block of Transmuter.

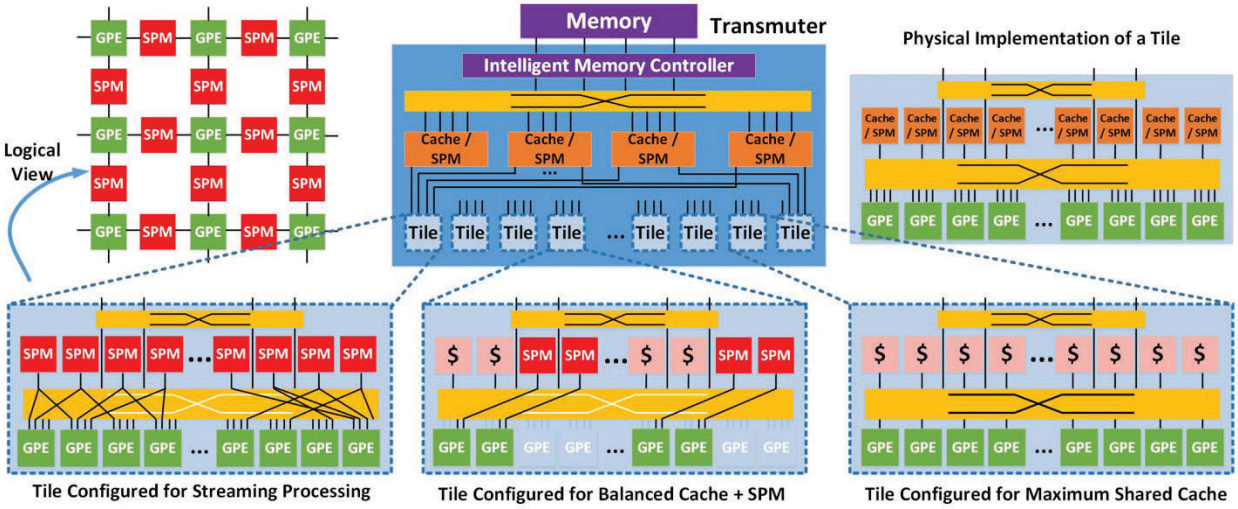


Figure 1: High-level Architectural Diagram of Transmuter Illustrating its Reconfigurability

2.1.1 General-purpose Processing Element and Local Control Processor

A GPE is an in-order Arm Cortex M-class processor, suitable for energy-efficient computation. It also has a small Silicon footprint, allowing Transmuter to incorporate many GPEs in present-day reticle sizes. The GPEs support the Arm v7 Thumb ISA without SIMD instructions and contain a single-precision Floating-Point Unit (FPU). Minor modifications are made to the GPE pipelines to handle control hazards introduced due to a custom PUSH/POP interface. A subset of GPEs connected to the first-level of Transmuter is termed a *tile*. The GPEs within a tile are coordinated by a control processor, the Local Control Processor (LCP). Each LCP has a private data cache that connects to main memory.

2.1.2 Work/Status Queues

The LCP in a tile distributes work to the GPEs in the form of 32-bit packets through hardware FIFO work queues that are private to each GPE. A GPE can similarly publish its status, e.g.

when it has finished a piece of work, via private status queues that connect to the TM through an arbiter.

The queues can be pushed to/popped from using loads/stores. Low-overhead decode logic translates incoming `ldr` instructions as POP and `str` instructions as PUSH commands, respectively. An important technique employed in this work to curtail dynamic power consumption in the TM and GPEs is to block the GPE/TM pipelines if there are structural hazards in the work/status queues. Specifically, if a work/status queue is empty and the core attempts a POP, the response only returns when a producer core pushes into the queue, and vice Transmuter v1.0 for a PUSH access. The same strategy is used for systolic array mode accesses.

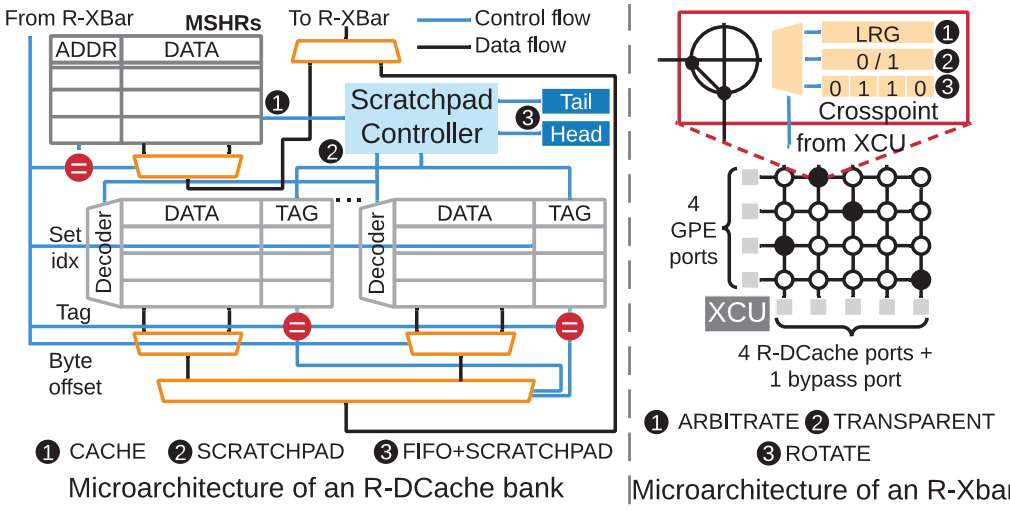


Figure 2: Microarchitectures of a Reconfigurable Data Cache (R-DCache) (left) and a Crossbar (R-XBar) (right)

2.1.3 Reconfigurable Cache

Transmuter has two layers of reconfigurable data cache (R-DCache) banks, the L1 and the L2 (Figure 2 – left). Each layer consists of an array of SRAM memories attached to controllers. They can operate in each of the following modes:

• **CACHE.** Each memory bank is accessed as a non-blocking, write-back data cache bank with LRU replacement policy.

• **SCRATCHPAD**. The tag array and Miss-Status Holding Registers (MSHRs) are powered off and the bank can be used as software-managed SPM.

• **FIFO+SCRATCHPAD**. The banks are utilized as SPMs with a programmable partition addressable as a FIFO queue, through a set of 32-bit head/tail pointers. Physically, these exist as registers next to the SRAM array. Further, the depth of the queue can be configured through memory-mapped registers. This mode inherits the single-instruction PUSH/POP capabilities of the work/status queues.

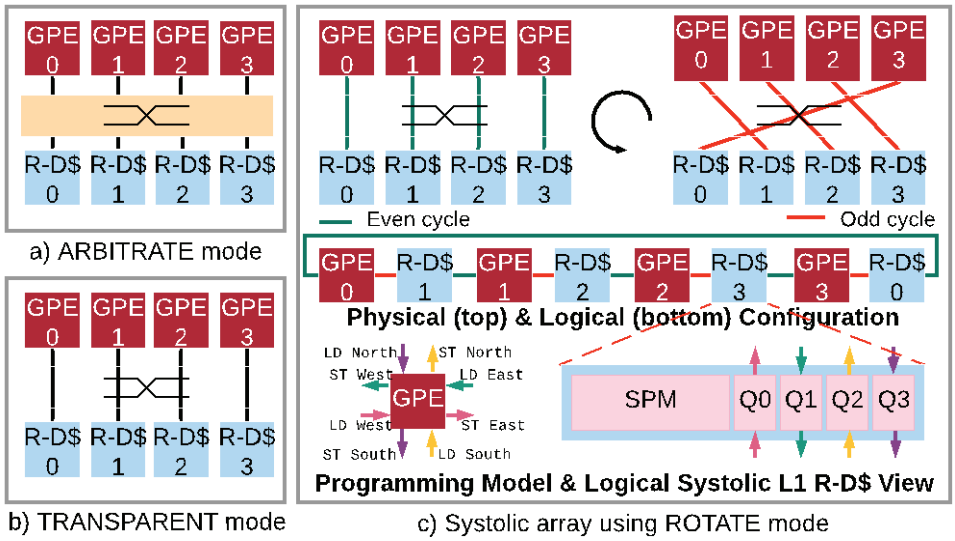


Figure 3: Illustration of the Three Supported Modes for the Reconfigurable Crossbar (R-XBar)

2.1.4 Reconfigurable Crossbar

An $M \times N$ crossbar allows M requesters access to N resources. The implementation of a reconfigurable crossbar (R-XBar) in Transmuter is illustrated in Figure 2 (right). The crosspoints can be programmed to arbitrate between the requesters, sup- port fixed connections (ON/OFF), or rotate through a series of ON/OFF patterns defined by shift registers at each cross- point. A small control block, the Crosspoint Control Unit (XCU), is used to program the crosspoint based on the mode, as memory-mapped I/O. The crossbars in Transmuter support the following configuration modes (Figure 3):

• **ARBITRATE.** When multiple requesters may attempt to access the same resource, the requests get serialized and priority is handled using a Least-Recently Granted (LRG) policy. One cycle is spent for arbitration.

• **TRANSPARENT.** Requester *i* has direct access to its corresponding resource *i*. Within a tile (L1), a GPE can access adjacent (private) memory banks with no arbitration penalty. For L2, each tile has exclusive access to one bank.

• **ROTATE.** The crossbar port connections cycle through a set of pre-programmed patterns that connect a requester and a resource. Figure 3 (c) shows how this configuration is used to emulate a 1D systolic array using 2 patterns (even and odd). This is extended to 4 patterns to compose a 2D systolic array. The programming model for this mode is discussed later. The GPEs can access the corresponding memory banks with no arbitration delay.

In addition to the higher L1 R-XBar layer for GPE↔ SPM communication, a lower R-XBar layer amplifies on- chip bandwidth between the L1-RCache banks and the L2. Each L1 and L2 R-XBar has an additional “bypass” port to allow GPEs to communicate to main memory when the corresponding R-DCaches operate as SPMs. The L1 crossbars in the proposed Transmuter

design have a bus width of 64 bits (32 address + 32 data bits) in each direction. L2 crossbars are wider (32 address + 128 data bits = 160 bits), as they transfer entire cachelines in bursts.

2.1.5 Synchronization Scratchpad

Transmuter averts overhead associated with writeback and broadcast messages by implementing coherence in software, as with prior accelerator designs [1,2]. Transmuter has a small global SPM, the Synchronization SPM, exclusively for synchronization operations that allow for implementation of software coherence and standard primitives such as locks, condition variables, barriers and semaphores. The interface between the synchronization SPM and the TM/GPEs is constructed using a low-throughput two-level arbiter tree. Accesses to this SPM were not observed to be a bottleneck in any of the benchmarks evaluated.

2.1.6 Miscellaneous Hardware

In addition to the hardware blocks discussed thus far, Transmuter consists of private instruction caches for each GPE and TM, to support a MIMD paradigm with different cores running independent code. MIMD support is useful for applications composed of many independent kernels, such as wireless communication applications [3]. The ICaches share access to the main memory with the L2 datapath through a two-level arbiter tree, not shown in Figure 1. Each GPE is augmented with a combinational block to route packets to a work/status queue, synchronization scratchpad, L1 R-DCache or the lower R-XBar in L1, based on a set of base-and-bound registers. Each TM has a similar block to decode between cache, scratchpad and work/status queue accesses.

2.2 Reconfigurable Features

This section details the overhead of dynamic reconfiguration, followed by various configurations that Transmuter can transform into and the software support. A shorthand notation of an $M \times N$ Transmuter system is used in the rest of the document, where M is the number of tiles and also the number of L2 R-DCache banks per tile, while N is the number of GPEs per tile and number of L1 R-DCache banks per tile.

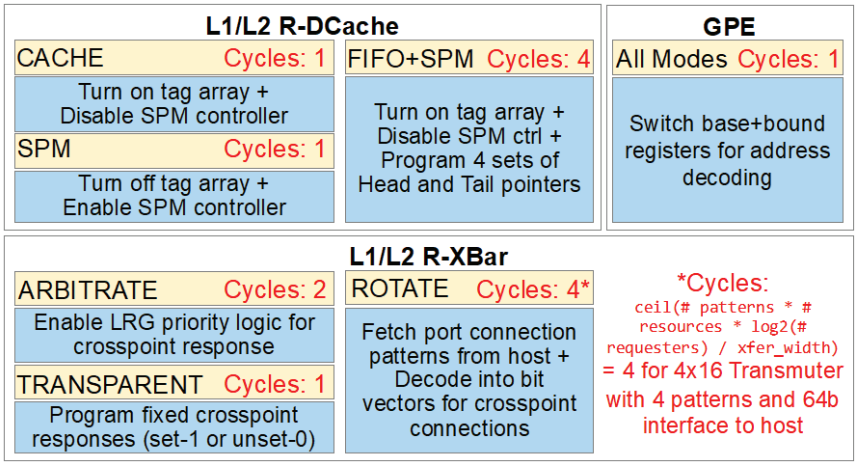


Figure 4: Steps involved in dynamic reconfiguration of 4x16 Transmuter
Reconfiguration is initiated for each Transmuter component in parallel. The hardware reconfiguration cost is shown in red, with R-XBar in the critical path.

2.2.1 Runtime Overhead

Transmuter can be reconfigured dynamically within a few cycles. Figure 4 shows the steps involved in Transmuter reconfiguration for each mode. The reconfiguration is initiated by an external host that sends Transmuter a command packet with relevant metadata as pay- load. This overhead combined with the values in Figure 4 leads to a reconfiguration time of <10 cycles for a 4×16 Transmuter implementation.

2.2.2 Cache + Crossbar Modes

Each of the Transmuter modes are presented here, along with a discussion of the API that exposes these modes to the programmer, abstracting away the low-level hardware specifics. Also discussed are use cases that can help a programmer determine the modes that best suit the compute/data access patterns of a given kernel.

The L1 and L2 R-DCache layers (Figure 4) can be set to one of the following modes. For each of these, the corresponding L1 R-XBar is set to ARBITRATE mode.

2.2.2.1 Unified Shared Cache

All GPEs within a tile (in case of L1), or all tiles (in case of L2) view the R-DCache banks as a unified shared cache. In this configuration, caches must be flushed after each "phase" of computation is finished, e.g. after the result matrix has been produced in case of GeMM, due to the absence of hardware coherence support. In addition, for shared L1 cache, tiles need to work on disjoint cachelines to ensure functional correct- ness of a program. For regular workloads such as GeMM, the scheduling is trivial. For other workloads, the synchronization SPM can be explicitly used to implement low-overhead coherence.

API Calls. The API call CACHE_FLUSH(), exposed to the programmer, triggers a flush of all the caches in the system.

Use Cases. Workloads exhibiting high temporal reuse as well as reuse between GPEs, similar to multi-core CPUs.

2.2.2.2 Unified Shared Scratchpad

In this case, all the banks in the layer function as a unified multi-banked SPM. It has a dedicated address space that is a disjoint slice of the global memory space. The SPMs are programmed to be inter- leaved at word-granularity, but can easily be reconfigured to byte, half-word or double-word granularities.

API Calls. The SPM_LOAD(…) and SPM_STORE(…) function calls present an illusion of a shared SPM space to the programmer; more specifically, a shared address space across all GPEs within a tile in case of L1 and across all tiles for L2. Further, since the SPM address map is dependent on the L1/L2 mode, GET_L1_SPM_TOP(...) and GET_L2_SPM_TOP(...) are used to return a pointer to the last address of the SPM space in L1 and L2, respectively.

Use Cases. Workloads that show low reuse or involve data better managed explicitly based on the programmer's expertise and where the data is shared across GPEs.

2.2.2.3 Private Cache

Each bank of the R-DCache layer is privately mapped to its corresponding GPE. In accordance with the previous section, the absence of coherence in Transmuter implies that GPEs working with private caches need to work on independent cachelines. Similarly, for the L2 in private cache mode, each tile must touch disjoint cachelines to ensure program correctness.

API Calls. The CACHE_FLUSH() API discussed previously is useful in this mode of operation to flush out privately written dirty data to the main memory.

Use Cases. Parallel workloads that naturally work on independent data streams, which are unpredictable and better managed by hardware caching.

2.2.2.4 Private Scratchpad

An R-DCache bank is privately accessed by its corresponding GPE in this mode, through a dedicated address space. Each GPE views the same logical SPM space as a subset of the main memory space, but all accesses physically are routed to the private SPM bank.

API Calls. The APIs for scratchpad access discussed in the previous section are also available for this mode.

Use Cases. Workloads consisting of independent data streams that exhibit low reuse and data prone to eviction/thrashing, or to store a part of the program stack for fast access to local variables within a function call.

2.2.2.5 Systolic Array

Systolic arrays are data-parallel architectures that consist of multiple processing elements, each of which receives data from its neighbor, performs a small amount of computation, and passes it to its next neighbor (Kung, 1982). If tuned (pipelined) correctly, this form of data orchestration exploits the largest possible degree of parallelism.

Transmuter can be reconfigured to mimic a "macro" 1D and 2D systolic array configuration, where each element of the array is a GPE that performs bulk computation before exchanging data with its neighbor(s). The R-XBars are programmed in ROTATE mode with patterns that switch the GPE ↔ SPM connections at a programmable number of cycles. Figure 3 (c) illustrates the use of ROTATE to allow fast access to the SPM banks, parts of which act as buffers between the systolic array GPEs. The ports of the crossbar are switched periodically, allowing multiplexed access to the buffers. The GPEs still retain access to the remainder of the SPM bank.

API Calls. PUSH_TO_GPE(...) and POP_FROM_GPE(...) are highly abstracted calls that accept a direction parameter (North, South, East or West). Internally, these translate into loads and stores to special memory-mapped addresses. These requests are handled by the XCU in the crossbars and routed (without arbitration) to the corresponding memory bank. This is dictated by the ROTATE configuration bits, as discussed previously. For the ease of programmability, these APIs can be used for both intratile, as well as inter-tile communication, with the routing logic handled in hardware.

2.3 Register-to-Register Tunneling (R2R)

Register-to-register tunneling (R2R) is an architectural enhancement to Transmuter's original systolic array mode that reconfigured the L1 and L2 memories as buffers to move data across cores. R2R instead uses the cores' register files to exchange data, thus avoiding expensive SRAM access in terms of energy compared to small registers. Figure 5 below shows a block diagram of the modifications made to the core design to support this feature.

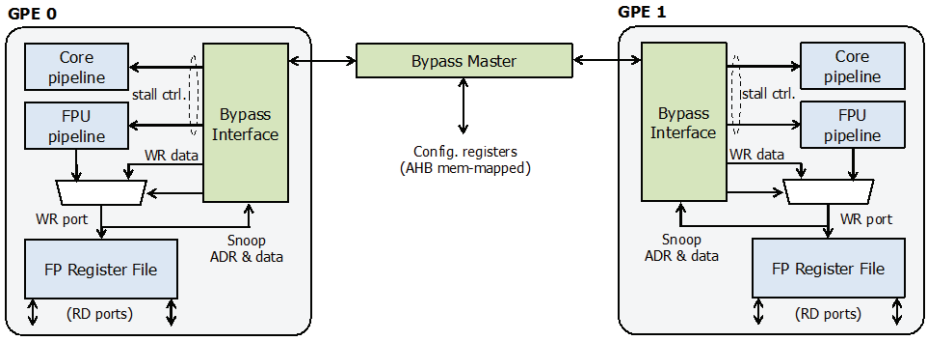


Figure 5: Accuracy and Performance of Our Trace-based Simulator

While efforts on R2R in Phase 1 were focused on early RTL development and validation of modifications to the Cortex-M4F cores, work in Phase 2 emphasized broader software support, and hardware optimization prior to tapeout. In particular, we incorporated a full set of R2R APIs into the Transmuter software libraries, optimized the R2R RTL for physical design to achieve higher

FMax, and rigorously validated the R2R-enhanced design using a suite of parallelized algorithms.

2.3.1 API and gem5 Integration

The R2R architectural feature has been fully integrated into gem5. There are two supported R2R configurations – R2R-basic that performs R2R pushes and pops using mov instructions, and R2R-opt that directly outputs or inputs operand values from the neighboring core's register files. The code snippet below shows an example of a GPE executing FIR filtering using R2R in the systolic mode.

```
for (int i = 0; i < N; ++i) {              // Iterate over each data value
    float x_ti = GPEQ_POP_FLOAT();                 // Pop from work queue
    float pprod_ti = R2R_POP_EAST_FLOAT(); // Pop from east neighbor
    float res_ti = pprod_ti + x_ti * f;    // Compute MAC
    R2R_PUSH_WEST_FLOAT(res_ti);           // Push MAC result west
```

2.4 In-Memory Transposition for Sparse Linear Algebra

We developed an in-memory accelerator that optimizes sparse matrix transposition to assist sparse linear algebra applications. The design places a custom accelerator in the DRAM buffer chip to eliminate the memory interface bottleneck and expose the high internal bandwidth while introducing minor modifications to commodity DRAM devices. The accelerator features a very wide hardware multi-way merger coupled with buffer and control units that help make best use of the available memory bandwidth. Compared to two state-of-the-art implementations on CPU [4] and a sparse library implementation on GPU (cuSPARSE), the design achieved an average speedup of 19.1x, 12.0x, and 7.7x, respectively. The true efficacy of the proposed design lies in its improvements to end-to-end workloads and is showcased by integrating it to CoSPARSE, an intelligent graph framework based on Transmuter. The in-memory accelerator eliminates the need to store multiple copies of the input graph to support different dataflows, reducing the memory storage requirements by at least half, while adding only a 5% performance overhead in a case study experiment.

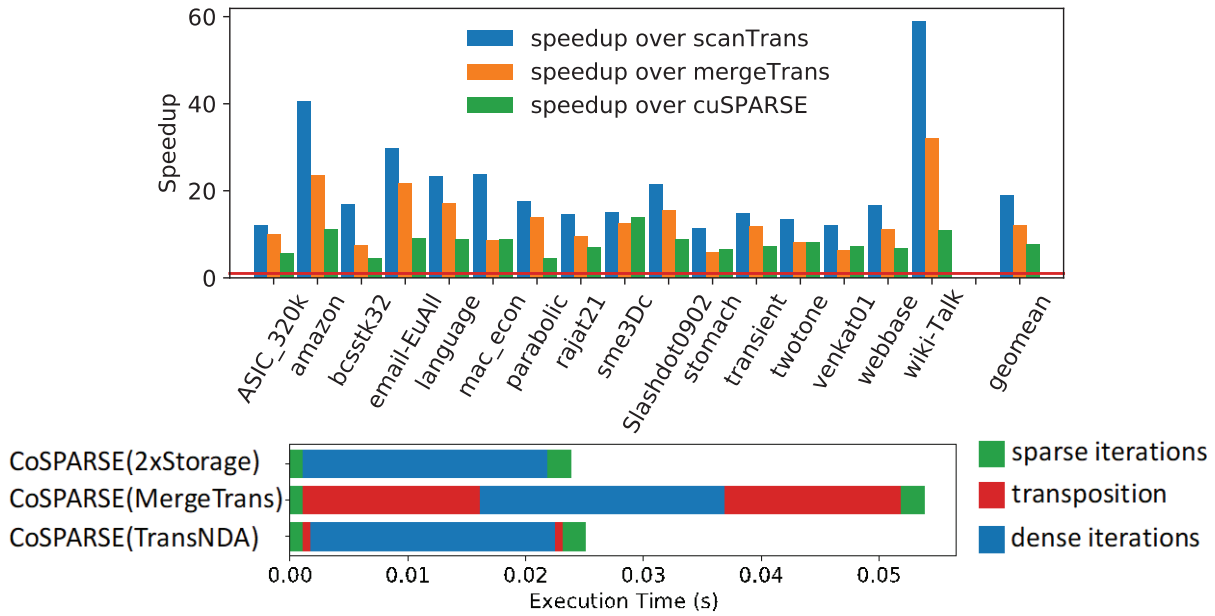


Figure 6. Top: Speedup of the Proposed Design Over Two State-of-the-Art Implementatins, scanTrans and mergeTrans [1], on CPU and the cuSPARSE Library on GPU
Bottom: The Execution Time of CoSPARSE Running SSSP Algorithm on the Graph amazon that (1) Avoids Runtime Transposition by Storing Multiple Copies of the Input Graph, (2) using mergeTrans [1] for Runtime Transposition, and (3) using the Proposed Design for Runtime Transposition

2.5 Hardware Validation Plan

Figure 7 shows the hardware validation plan for the Transmuter program. The OuterSPACE chip was presented in the first phase final report. In this final report we cover the Transmuter v1.0 chip design and testing, as well as the final prototype emulator which fully tests a complete host+transmuter system that is now ready to tape-out.

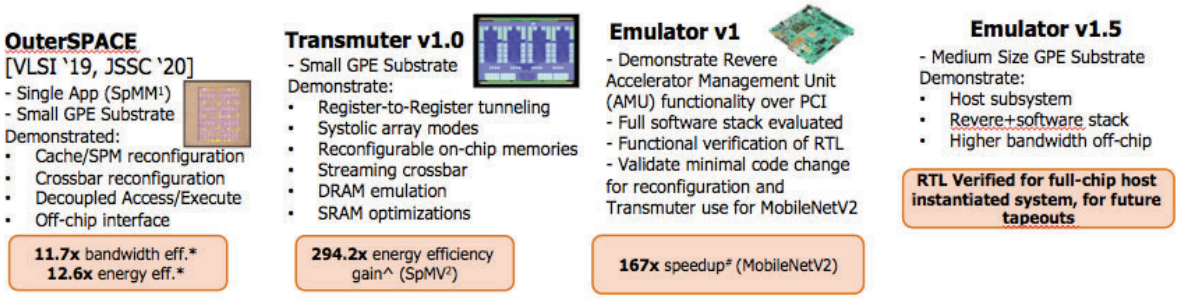


Figure 7: Hardware Validation Plan

2.6 Transmuter v1.0

We present Transmuter v1.0, an energy-efficient processor with 36 systolic ARM Cortex-M4F cores and a runtime-reconfigurable memory hierarchy. V1.0 exploits algorithm-specific characteristics in order to optimize bandwidth, access latency, and data reuse. Measured on a set of kernels with diverse data access, control, and synchronization characteristics, reconfiguration between different Transmuter v1.0 modes yields median energy-efficiency improvements of 11.6× and 37.2× over mobile CPU and GPU baselines, respectively. Conventional programmable architectures have structurally limited support for diverse, dynamically-varying algorithm characteristics. CPUs provide broad programmability, but are burdened by large out-of-order cores and costly data transfers through fixed-function, coherent caches. GPUs have large SIMD thread arrays and employ scratchpad memories but suffer on irregular workloads. FPGAs are highly configurable but incur high hardware overheads and long reconfiguration times. In contrast, Transmuter v1.0 is a highly-parallel architecture that supports regular, irregular, and systolic workloads via fast (2 cycle) reconfiguration. The flexible architecture enables algorithm-specific runtime optimizations. Finally, a tree-based method accelerates multi-threaded barrier synchronization operations.

2.6.1 Architecture Overview

The Transmuter v1.0 architecture (Fig. 1) consists of compute tiles and a 3-level memory hierarchy. Each tile has a cluster of 8 ARM Cortex-M4F worker cores equipped with IEEE 754-compliant floating point units (FPUs). We introduce register-to-register (R2R) links between workers in a 4x2 spatial array, extending transparently across tiles, to support chip-wide systolic algorithms with efficiency gains up to 66.8×. A reconfigurable crossbar (RXB) and 8-slice reconfigurable on-chip memory (ROCM) provide a number of data transfer patterns, each optimized for access latency and throughput. Fast access to cross-mode persistent data (e.g., mutexes) is supported with fixed-function scratchpads in each tile and at the global level. Supervisory tasks and runtime reconfiguration are offloaded to a manager core in each tile.

2.6.2 Reconfigurable Crossbar and Memory

The RXB and ROCM (Fig. 4) are co-designed to provide reconfigurable memory slices that are private per worker, shared across all workers in a tile, or FIFO-buffered between core-pairs. The RXB has 1 bidirectional port per worker/ROCM slice and is reconfigurable to either 1) *RXB-shared*, 2) *RXB-private*, or 3) *RXB-queue* modes. RXB-shared provides all-to-all connectivity between workers and ROCM slices with least-recently-granted arbitration [cite]. From the perspective of worker cores, shared slices function as single memory that has 8x the capacity of private slices. RXB-shared is also beneficial for workloads where common data is accessed by multiple cores. In RXB-private, RXB crosspoints lock worker-to-slice connections vertically and skip arbitration cycles, reducing access latency by 33%. Privatization eliminates false bank contention with up to ~8× additional improvement in latency and bandwidth. RXB-queue supports common streaming DSP and filtering kernels. Since addresses are unnecessary for streaming FIFO accesses, 'crosspoint splitting' enables simultaneous reads/writes by producer-consumer cores over the same RXB port, and effectively doubled bandwidth.

The L1 ROCM is reconfigurable into cache, SPM, or queue (i.e., FIFO) modes. Cache logic with tags and hit/miss detection is conditionally enabled and data-gated if unused. SRAM banks are fully reused across modes, along with multi-purpose registers that track mode-specific request states (e.g., cache miss handling, FIFO fill levels). 32-bit wide sub-banks match the native bit-width of worker cores. Sub-banking trades 34% area increase for 3.4× lower common-case access energy. RXB-shared and private combine with ROCM cache or SPM, resulting in 2x2 composite configurations in addition to FIFO queue. Mode control is memory-mapped to the manager core, and mode transitions complete in 2 cycles.

2.6.3 R2R Tunneling

R2R forms a chip-wide, systolic array configuration where adjacent cores communicate directly, avoiding crossbar and cache overheads (Fig. 2). If enabled at runtime (i.e., in software), the FPU registers s0-s3 are aliased to scalar data links in the <W, E, N, S> directions, respectively. When an instruction writes to an R2R register, data from register writeback - normally directed to the local register file - is instead intercepted by the R2R Shim, and forwarded to an adjacent core. An R2R systolic-write updates the link state, and allows a matching R2R systolic-read to proceed at the neighbor. Systolic-reads from R2R registers proceed normally if the link state is valid. Stall-based flow control prevents stale reads and destructive writes. R2R is tightly-coupled with the M4F core, and flow control is implemented with virtually no overhead by integrating with pipeline stall mechanisms. Similarly, link state (i.e., data valid tracking) requires only 2 bits per link.

2.6.4 Tree-Based Scratchpad Barriers

Synchronization barriers are key operations in multi-threaded programs that consume up to 70% of cycles for complex workloads [cite]. Transmuter v1.0 averts coherence overheads with dedicated scratchpads that provide predictable low-latency access. To reduce the serialized barrier section, scratchpads are placed at the tile (T-SPM) and global (G-SPM) levels, enabling a tree-based strategy [cite]. The centralized scratchpad-based approach alone achieves a 1.7× speedup compared to cache-based barriers from the pthreads library on CPU. The tree-based strategy yields additional 3.8× speedup (6.5× total), despite a 9× higher threadcount.

2.6.5 Measurement Results

Transmuter v1.0 was tested on MachSuite kernels [cite] against a mobile-class 4-core ARM A57 CPU and 256-core Tegra X1 GPU. Stencil2D (2D convolution) and GeMM (matrix mult.) exhibit regular data accesses to dense data, while KMP (string search) and SpMV (sparse matrix-vector mult.) have data-dependent variation in access patterns. Mergesort is a branch and synchronization-heavy comparison-based sort. We selected 2 Transmuter v1.0 modes per kernel, and modulated data sizes up to 512 KB.

Across kernels, median energy-efficiency improvements of 11.6× and 37.2× are achieved versus the CPU and GPU, respectively. Energy-efficiency improvements between Transmuter v1.0 modes extend up to 3.17×, illustrating how reconfiguration captures workload-dependent variation. For instance, Stencil2D with Transmuter v1.0 private cache yields 1.37× higher GFLOPS/W relative to private SPM+R2R at small data sizes, but the advantage between modes

is inverted at larger sizes. This result is largely due to the use of R2R to share and reuse overlapped input patches across cores, and cache pressure as dataset footprint increases. On Mergesort, Transmuter v1.0 attains 2.33× and 71.6× speedups over the CPU and GPU, respectively, translating to 14.4× and 105× energy-efficiency improvements. GPU profiling indicates bottlenecks in parallel synchronization and branch-heavy comparison operations. Results from Mergesort suggest that Transmuter v1.0's independent scalar cores and tree-based scratchpad barriers are effective in-practice for irregular kernels.

Transmuter v1.0 was fabricated in 28 nm CMOS and occupies 12 mm^2. At 1.0V nominal voltage, the chip operates at 510 MHz clock frequency resulting in 11.9 GFLOPS peak-practical performance and 810 mW power dissipation. Voltage scaling to the 0.6V minimum-energy point (MEP) improves energy-efficiency by 2.47×
(36.4 GFLOPS/W) while dissipating 7.9 mW at 31 MHz. Energy-per-cycle varies from 543 - 1588 pJ/cycle between 0.6 - 1.0V.

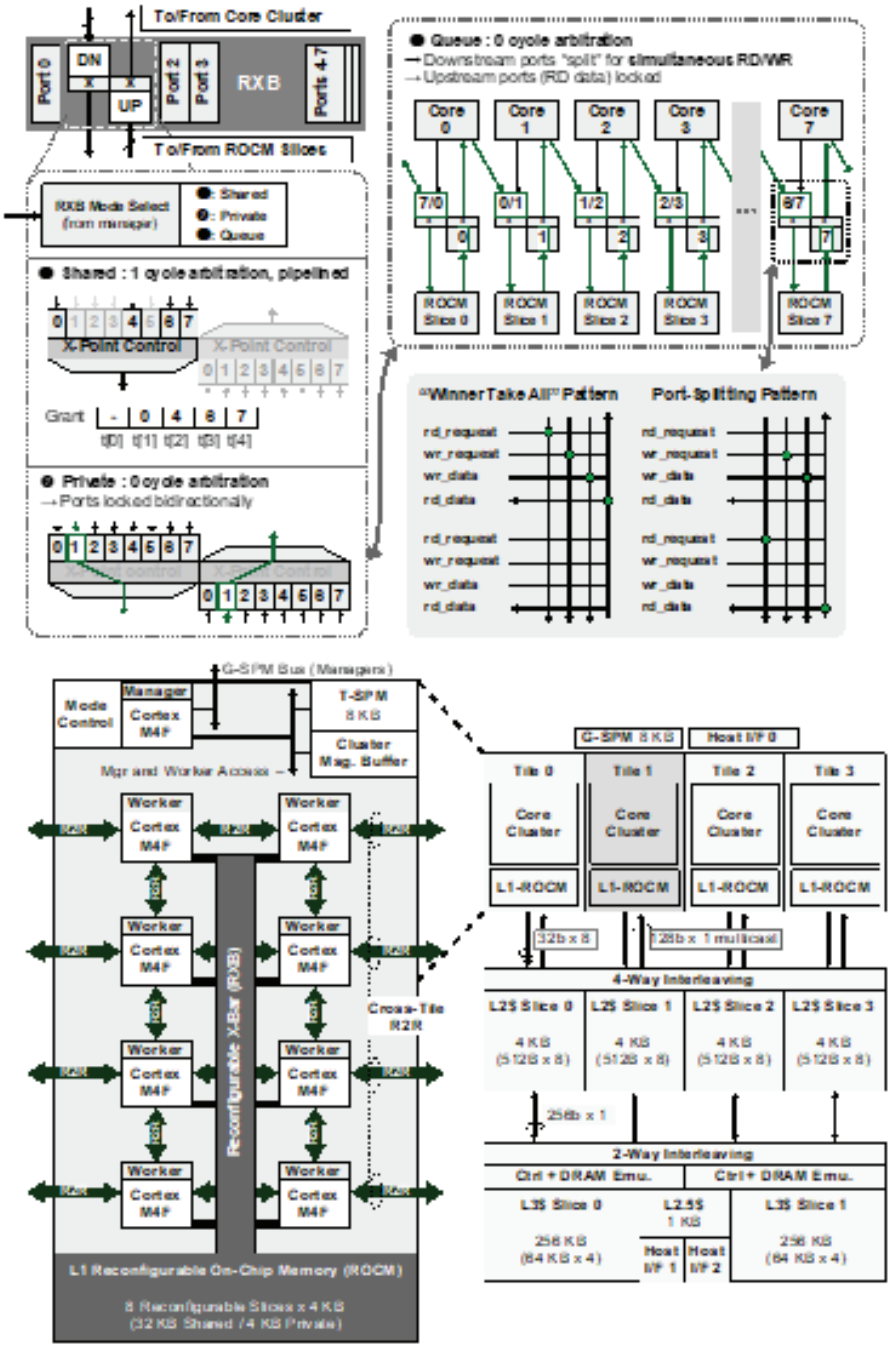


Figure 8: Transmuter

FPGA Emulation – Tranmuter v1.5

The hardware team worked on SDH emulation on the HTG960 FPGA board. This effort required changes from the SDH ASIC design to a modified FPGA-target design harnessed into a single design. Due to the limited FPGA size on the HTG960 (biggest FPGA commercially available), the ported design has a reduced transmuter configuration from 16 x 8 workers to 8 x 8 workers. The host subsystem consists of a single core A32 Processor. The design has updated kernels and

RTL configurations but clock ratios are preserved compared to ASIC so that the performance can be measurements in clock cycles.

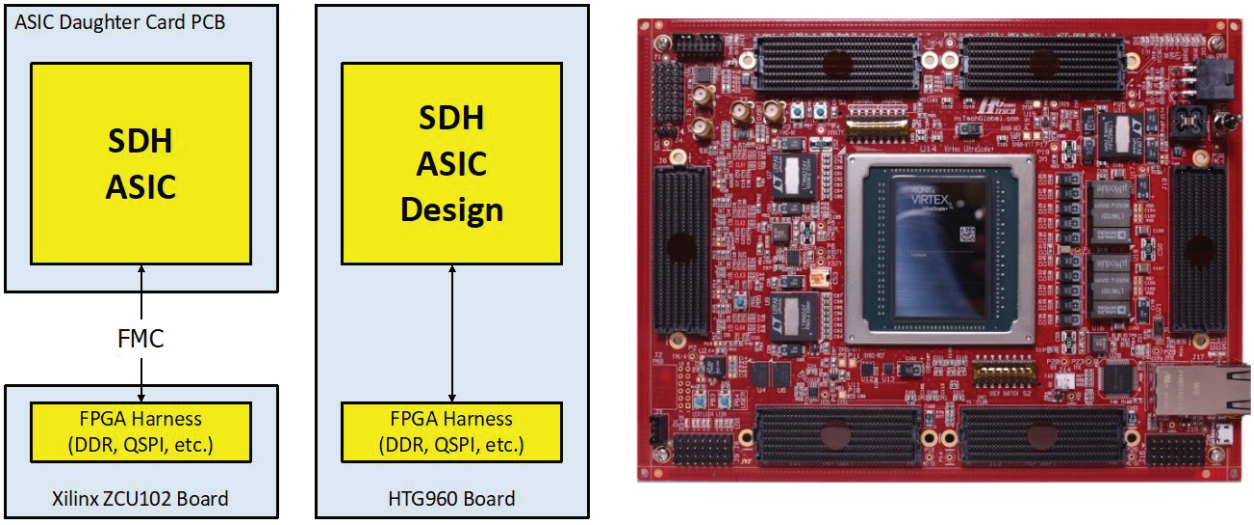


Figure 9: FPGA Emulation

The team has accomplished porting and validating the RTL functionality on the HTG960 Board. The team also validated Transmuter RTL updates and kernels on a smaller scale without host subsystem. Tested software APIs and kernels (multiple variants) include: hello_world, point_prod, stencil2d, kmp, gemm, mergesort, and spmv_crs.

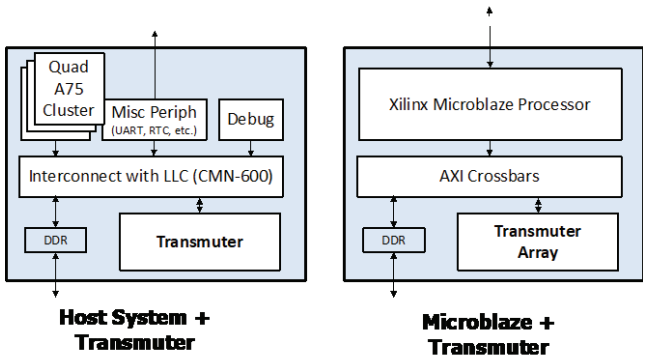


Figure 10: Host System + Transmuter and Microblaze + Transmuter

2.7 Technology Transition

Multiple PIs in our SDH program have been participating in the DARPA DSSoC DASH project as a performer. For technology transition, we are collaborating with a startup company DASH Tech IC which is a spin off from the DSSoC DASH project. The figure below provides an overview of the DSSoC DASH project. DASH is a DARPA ERI project launched at the same time with Transmuter SDH. Arizona State University is the lead organization for DASH. UM leads the hardware SoC design effort and other performers include University of Wisconsin, University of Arizona, Arm, General Dynamics, and Collins Aerospace. The main goal of DASH is to

design a new SOC with ASIC-like performance and CPU-like programmability for the DoD wireless application domain.

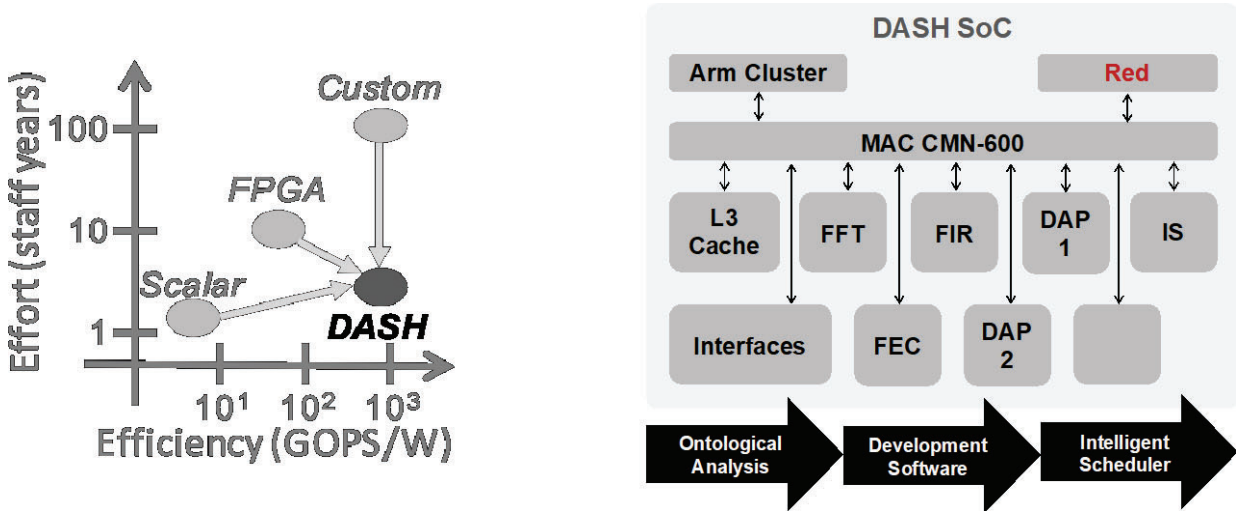


Figure 11: DASH effort and SoC

DASH SoC is designed to address a wide range of DoD and commercial wireless applications. Because DoD wireless applications use proprietary protocols for secure and robust communications, commercial ASIC solutions are not really usable for the majority of DoD applications. Many DoD wireless communications and spectrum sensing applications rely on FPGAs as they provide significantly better performance than general purpose processors. But FPGAs are still very power hungry and their performance is not as high as ASICs. Another problem is that the reprogramming time for FPGA is slow.

The DASH SoC is designed to replace FPGAs so that DoD wireless applications can benefit from ASIC-like performance and CPU-like full programmability. The spinoff startup company DASH Tech IC has received funding from the EEI program and several DARPA programs such as space BACN and Prowess. We are working with DASH Tech IC as a potential tech transfer partner to integrate Transmuter in the DASH SoC for DoD wireless applications.

The key enabling module in the DASH SoC is a new processor called Domain Adaptive Processor or DAP. The figure below shows a high level overview of DAP, which is a fast programmable systolic array fabric of heterogeneous processing element cores. Its custom ISA is specifically designed for reduced programing overhead for systolic streaming kernels that are widely used in wireless applications. It has a very efficient loop controller and it does not have an instruction decoder so that the compute and data movement overhead is minimized in DAP. Thus, it can achieve ASIC like performance and efficiency for streaming systolic kernels.

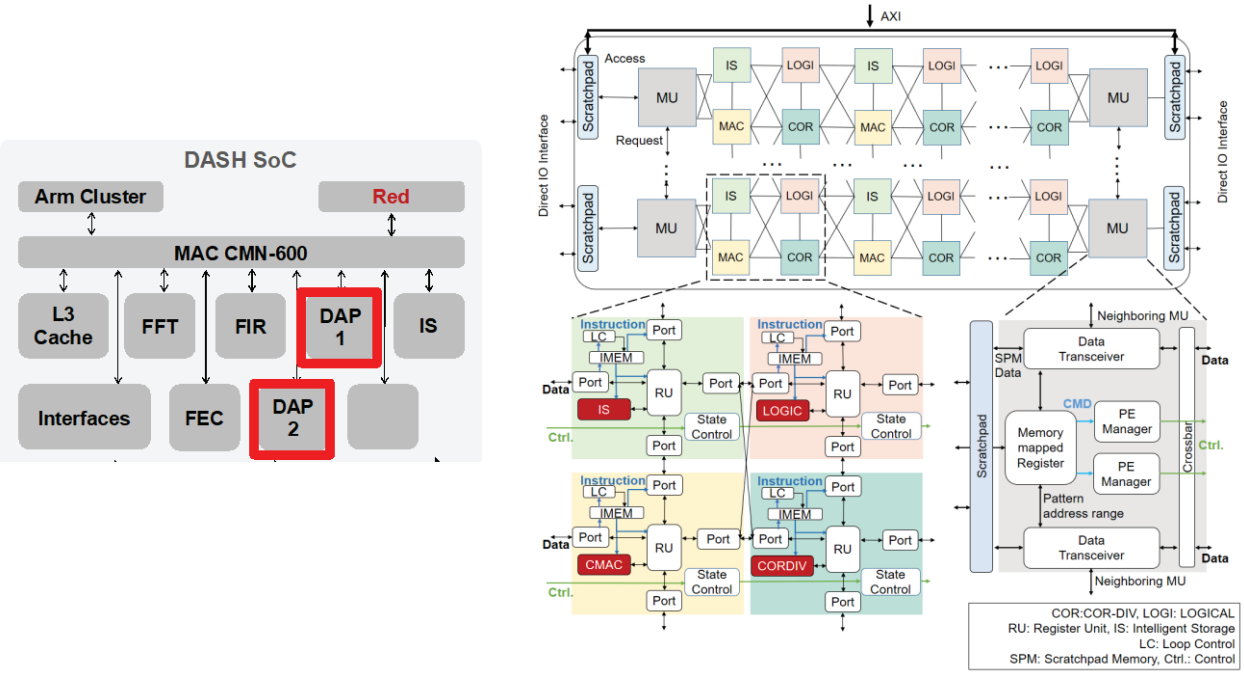


Figure 12: High level overview of DAP

The table below shows how Transmuter and DAP can be complementary to each other. DAP has higher compute density of 256 cores per 21 mm^2, which is higher than that of Transmuter which has 128 cores in 16mm^2 area. The compute density difference come from that DAP uses compact custom designed PEs but Transmuter uses general purpose Arm Cortex M4F cores.

Since Transmuter uses Arm ISA, programing can be done using standard C++ language and Arm compliers. On the contrary, DAP programming support is very limited and we are currently relying on a custom assembler, which makes DAP programming quite difficult. When the application requires complicated dataflow, Transmuter programming and debugging would be much more convenient than programming DAP. In terms of architecture, Transmuter can be on-the-fly reconfigurable with R2R tunneling with systolic mode and it can also support non-systolic cache mode data path. However, DAP only supports the systolic mode for streaming kernels without branching operations. When the application requires data-dependent control, Transmuter can be a much more powerful option and DAP.

One of the main advantages of Transmuter over DAP is floating point operations. Transmuter supports 32-bit floating point operation while DAP has a limited precision of 16-bit fixed point. DAP has substantially higher computation efficiency and performance than Transmuter for fixed point streaming systolic kernels. But direct comparison for energy efficiency is unfair because the Transmuter efficiency number is from floating point operations on non-systolic kernels.

Table 1: Transmuter and DAP

	Transmuter	DAP
Core	**Arm Cortex M4F**	**Custom PE**
Compute Density (GF 12nm)	128 cores / 16.25 mm^2	**256 cores / 21.16 mm^2**
ISA / Compiler	**Arm ISA + Complier**	Custom VLIW + assembler
Architecture	**Reconfigurable core array with R2R tunneling**	Systolic array
Precision	**32-bit floating point**	**16-bit fixed point**
Energy efficiency	15 – 36 GFLOPs/W	**Up to 1000 GOPs/W**

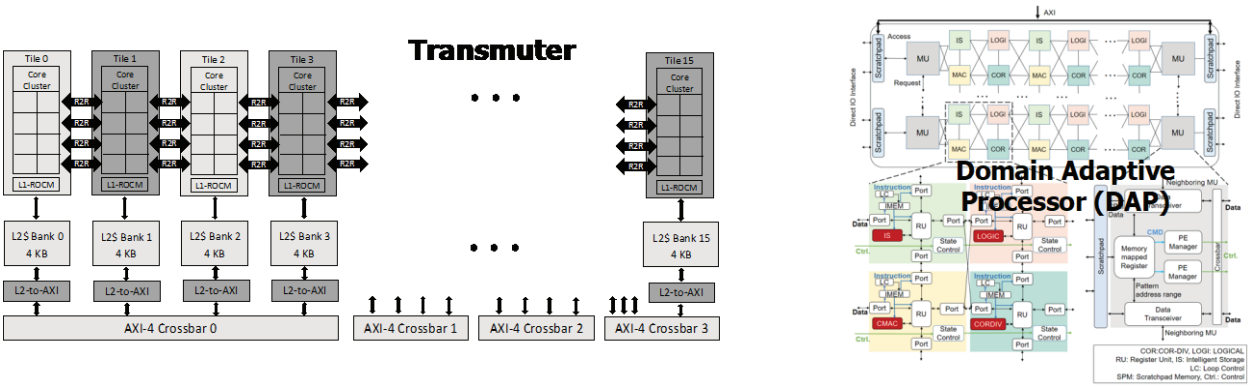


Figure 13: Transmuter and DAP

For tech transition, the PIs are working together with DASH Tech IC to validate the concept of a new SoC that integrates Transmuter in the SoC together with DAP. The diagram below shows the SoC concept that contains both Transmuter and DAP. This architecture is essentially identical to our SDH SoC except that it has DAP and additional accelerators available on DASH SoC. The Arm host subsystem is identical between DASH and SDH.

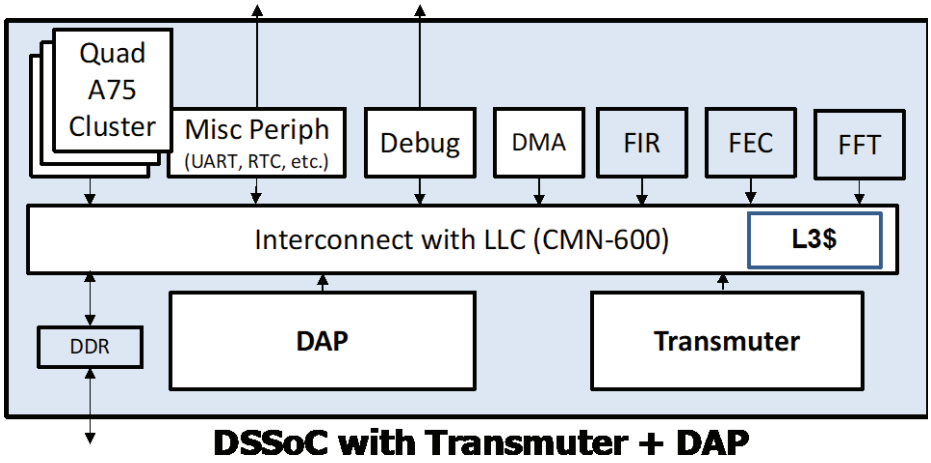


Figure 14: DSSoC with Transmuter and DAP

Transmuter can complement DAP in this combined concept SoC in several aspects. We identified several DoD wireless communication and wideband spectrum sensing applications require floating point computations on Transmuter such as an 8-k point FFT, QR decomposition on an 80x80 matrix, large matrix-matrix multiplications and singular value decompositions for MIMO processing. DAP is not really suitable for those kernels since it has only 16-bit fixed point precision.

Another set of applications where DASH SoC would need Transmuter is when the application requires data-dependent control and non-systolic processing. DAP is great for systolic streaming kernels, but DAP can only handle deterministic streaming datapath without any branching operations. On the other hand, Transmuter can support not only systolic kernels but also non-systolic sparse linear algebra kernels with data-dependent control.

Another recent trend in wireless applications is using more deep learning oriented approaches. DAP is good for DNN inference computations but it cannot execute on-chip neural network training due to limited precision and lack of non-linear function support. Transmuter can be useful for on-chip neural network training for future DoD wireless applications. We have also worked on porting large wireless application kernels onto Transmuter to quantify the performance of Transmuter for floating point large dimension wireless kernels.

3 SECTION III: ALGORITHIM ANALYSIS RESULTS

In this section we present our findings on the DARPA workloads as well as others of interest.

3.1 Performance Modeling

This work uses the gem5 cycle accurate simulator [5] to provide execution time and energy-efficiency (GOPS/W and GFLOPS/W) estimates on the Transmuter system. We started with a baseline multicore CPU model in gem5, simulated in Syscall Emulation (SE) mode. The SE mode simulates the bare metal hardware without including overheads for operating system initialization, switches from kernel space to application space, software thread management, etc. We adjusted the parameters (such as issue width, number of outstanding accesses, etc.) for the cores to closely model an M4 for both the GPEs and LCPs; these parameters are specified in the previous section. We then organized these cores into tiles and connected them to our two-level on-chip interconnected consisting of multi-banked caches and crossbars. The system is connected to a quad-channel DDR4 interface which is the main memory for the Transmuter system.

3.2 Power Modeling

The power consumption of Transmuter is estimated by evaluating the power for each component in the architecture. The power calculation is based on both the raw power parameters, i.e. unit static or dynamic power for a component or transaction energy per access, and the statistics, i.e. active cycles and access numbers, generated by gem5 simulation.

3.2.1 Power Estimation

The total power is computed as the sum of the static power and the dynamic power of each component. The static power is accumulated directly for all the hardware components. The dynamic power is calculated by taking the dynamic energy consumed by the workload and dividing it by the execution time resulting in the average dynamic power. The dynamic energy of cores is calculated by multiplying the dynamic power of each component with the number of active cycles. For the reconfigurable caches, the synchronization scratchpads, the reconfigurable crossbars, muxes and arbiters, and the memory controller, the dynamic energy is obtained by multiplying the transaction energy per access with the number of accesses to each component.

Technology Scaling

To estimate the power consumption of all components assuming a specific technology node, the raw power parameters are scaled using standard Dennard scaling. The static power uses quadratic scaling and the dynamic energy uses cubic scaling by the technology node. Since the frequency uses linear scaling, the resulting dynamic power will also be scaled quadratically.

Dynamic Voltage Frequency Scaling (DVFS)

In the DVFS model, the target voltage is calculated based on the formula $f \propto (V_{DD} - V_{th})^2/V_{DD}$, where f is the clock frequency, V_{DD} is the supply voltage, and V_{th} is the threshold voltage.
Given a target frequency f_{target} the target supply voltage is calculated based on the following equation:

$$\frac{f}{f_{target}} = \frac{\frac{(V_{DD}-V_{th})^2}{V_{DD}}}{\frac{(V_{target}-V_{th})^2}{V_{target}}} = \frac{(V_{DD}-V_{th})^2}{V_{DD}} \times \frac{V_{target}}{(V_{target}-V_{th})^2}$$

$$V_{target} = \begin{cases} V_{target}, & V_{target} \geq 1.3V_{th} \\ 1.3V_{th}, & Otherwise \end{cases}$$

The nominal supply voltage V_{DD}, nominal frequency f and the threshold voltage V_{th} are constants and are derived from measurements on a recently fabricated chip. The minimal target voltage allowed for correct functionality is set to be 30% higher than V_{th}. The target voltage V_{target} calculated is then used to scale down the total power of the system by $(V_{target}/V_{DD})^2$.

Original Sources of Unit Power

The power documented in the Arm Cortex M4F core specification is used to the GPEs and LCPs. For reconfigurable crossbars, muxes and arbiters, an RTL model based on a synthesizable version of swizzle-switch crossbar is synthesized to generate power estimation. The static power and transaction energy per access of the reconfigurable cache banks are modeled using CACTI 7.0. The memory controller uses the static power and dynamic energy per access of a standard DDR4 memory controller.

3.2.2 Calibration with Chip Measurements

To verify the accurateness of the power model, the power for a 4x8 transmuter system estimated by power model is compared against the power data obtained from the taped-out chip.

Cross-verification with chip measurements

The comparison between the power estimated by the original power model and the power measured from the chip is show in Table 2. Generally, the power model tends to be optimistic about the power consumption. The discrepancy between the estimated power and real power consumption exists because the power model over-estimates the static power and under-estimates the dynamic power. The inaccurate estimation of dynamic power also results in that the efficiency benefits achieved by applying DVFS on the fabricated chip is much more than what is observed based on results generated by gem5 simulation and the power model.

Table 2: Cross-verification with Chip Measurement

28nm, 1V, 800MHz, 4x8 Transmuter

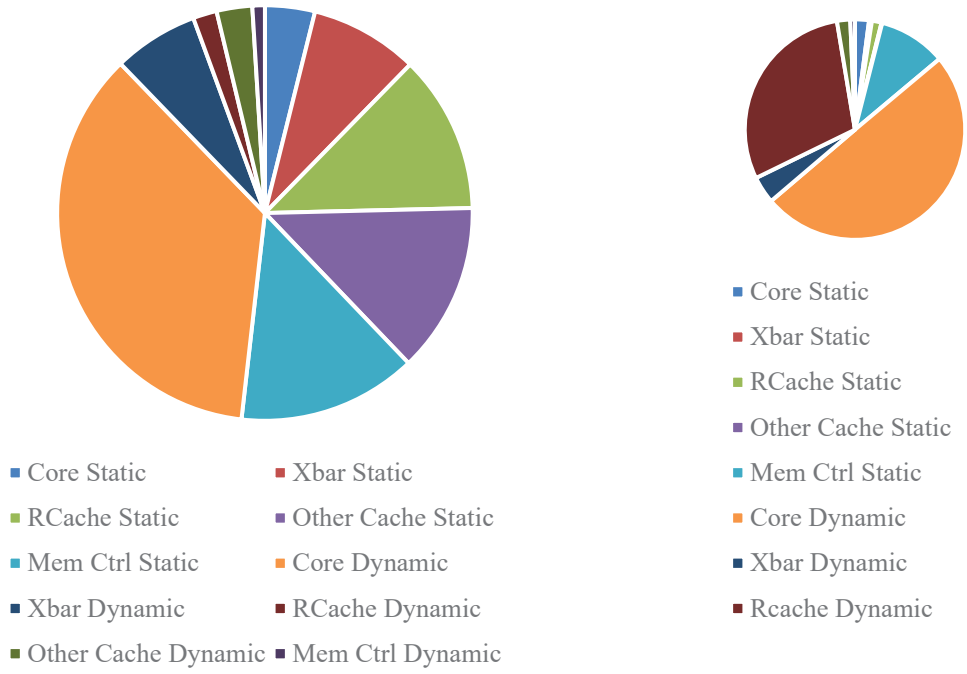
Workload	Original Power Model (W)	Calibrated Power Model (W)	Chip Measurement (W)
NOOP	0.41	1.06	1.11
SpMV	0.65	1.15	1.22
DMM	0.66	1.13	1.22

Adjustments to sources of unit power

We identified the main reason for the discrepancy between the power model and the chip power measurement to be the raw power parameters, especially the cache power data provided by the CACTI 7.0 cache model. Therefore, the raw power parameters in the power model are updated using the measured chip power and the detailed power breakdown provided by the synthesis reports. The estimated power generated by calibrated power model is shown in the Table 2. The calibrated power model is able to compute a much more realistic power estimation. Minor discrepancy exists between the estimated power and the chip measurements due to the difference in the off-chip interface between the chip and the modeled architecture.

Power breakdown by the updated power model

The original power model over-estimates the static power and under-estimates the dynamic power, especially for the reconfigurable caches. Figure 15 shows the power breakdown generated by the original and calibrated power model for GEMM. The static power computed by original power model takes up 52% of the total power. The static power of reconfigurable caches accounts for 24% of the total static power. The static power estimated by calibrated power model, instead, is around 10% of the total power, which is more accurate based on chip power measurements. The majority of the power is consumed by cores and reconfigurable caches, which are also the most active components in the architecture. The percentage of dynamic power of cores and reconfigurable caches now increased from 36% and 2% to 50% and 30% of the total power.



(a) Original power model **(b) Calibrated power model**

Figure 15: Power Breakdown Generated by the Original and Calibrated Power Model

3.3 Radar Workloads Overview

We implemented these four radar signal processing workloads in Transmuter and compared the performance with a baseline CPU implementation. The configurations of the two platforms are as follows:

- Transmuter: 4 tiles, 16 GPEs per tile, 4KB L1 and l2 cache banks, running at 1.0 GHz.
- CPU (Baseline): Intel Core i7-4510U, a dual-core CPU, 22nm technology, running at 3.1 GHz.

We summarize the comparison results in Table 3.

Table 3: Results Summary for 4 Radar Workloads

Workload	Time (s) $\left(\frac{TM-CPU}{CPU}\right)$	Power (W) $\left(\frac{TM-CPU}{CPU}\right)$	Energy (mJ) $\left(\frac{TM-CPU}{CPU}\right)$	GFLOPS/W $\left(\frac{TM}{CPU}\right)$
Pulse Doppler Radar	0.01498 **(-81.6%)**	0.413 **(-91.0%)**	6.186 **(-98.3%)**	2.67 **(60.7x)**
Radar Correlator	0.00038 **(-72.9%)**	0.152 **(-96.7%)**	0.058 **(-99.1%)**	2.92 **(111.5x)**
SAR Range Doppler Algorithm	0.04101 **(-34.1%)**	0.687 **(-87.1%)**	28.17 **(-91.5%)**	1.61 **(11.8x)**
SAR Backprojection	0.05821 **(-38.7%)**	0.700 **(-86.3%)**	40.75 **(-91.6%)**	5.80 **(11.9x)**

3.3.1 FFT Radix-2/4 implementation

All four radar workloads require different sizes of FFT kernels, which usually dominates the execution time. For higher efficiency, we implement FFT of size 512 in radix-2, and FFT of sizes 256 and 1024 in radix-4.

Radix-2 FFT Implementation. The radix-2 FFT algorithm computes Fourier Transform using $\log_2 N$ stages, where each stage consists of N/2 butterflies. It has a complexity of $N \log_2 N$. The implementation detail is shown in Figure 16 (left).

All GPEs in a tile process computations in a pipelined manner. All butterfly computations in one stage are computed by one GPE. The ping pong buffers between two GPEs are used to store the intermediate values between stages. Each GPE reads the input values, computes the top part of the butterflies in the stage (as indicated in the red circle), writes the output to the next buffer, and

computes the next set (green circle). Each GPE communicates with its neighbor via the R2R queue to make sure the latest data is ready before the next computation starts.

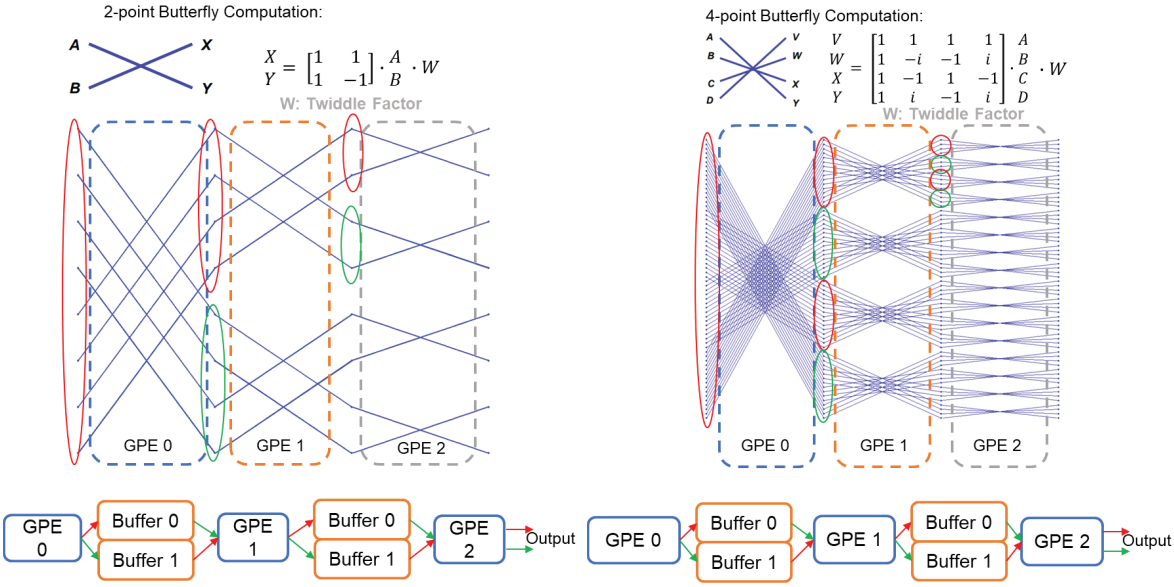


Figure 16: Pipelined Radix-2/Radix4 FFT Implementation

Radix-4 FFT Implementation. The radix-4 FFT algorithm computes Fourier Transform using $\log_4 N$ stages, where each stage consists of N/4 butterflies. It uses same number of adds ($N \log_2 N$) as Radix-2 FFT and 75% of multiplies ($\frac{3}{8} N \log_2 N$) of Radix-2 FFT. The implementation detail is shown in Figure 16(right). We follow the same pipelined implementation as radix-2 FFT. Each GPE reads the input values, computes the top branch of the butterflies in the stage (as indicated in the red circle), writes the output to the next buffer, and computes the next branch (green circle). After 4 branches of the butterflies in this stage are computed, the GPE loads new input values from the buffer and starts the new iteration.

Correctness. We evaluate the Mean-Square-Error (MSE) with the referenced FFT implementations in GSL library. The MSE is 0.0000.

3.4 Radar Correlator

Radar Correlator takes the received radar waveform and correlates it with the original transmitted waveform; the peak corresponds to an estimate of the range or time delay. The algorithm processes a signal of length 2 * *#sample* where *#sample* is the length of the pulse. Its main computation kernels are FFT-1024 and IFFT-1024. We implemented both these kernels using radix-4 FFT.

Flowchart and profiling results. The flowchart of Radar Correlator is shown in Figure 17. It computes the FFT of noisy signal inputs and the reference signal, multiplies them in the frequency domain and converts them back to the time domain. It then uses a *max* operation to extract the peak. We profile it on the CPU and find that *Xcorr* process dominates the time cost. The TM implementation shows a very similar trend as shown in Figure 18.

Simulation results. We implement radar correlator on TM and compare it with the CPU reference C implementation both in terms of energy efficiency (GFLOPS/W) and execution time (ms). The detailed results are shown in Table 4. We also provide the normalized results in Figure 19. Results show that TM running at 1.0 GHz takes 73% less execution time while achieving 97.3x higher energy efficiency compared to the CPU. TM running at 0.2 GHz takes approximately the same time compared to the CPU and achieves an even higher energy efficiency of 156.7x compared to the CPU.

Comparison with GPU. We further compare the performance with RTX 3090 GPU platform. The torch-based implementation has an execution time of 0.184 ms with a 0.065 GFLOPS/W as listed in Table 4. Compared to GPU, the time cost of TM at 1.0 GHzis almost twice at large but it has a significantly better energy efficiency – higher by 41.7x.

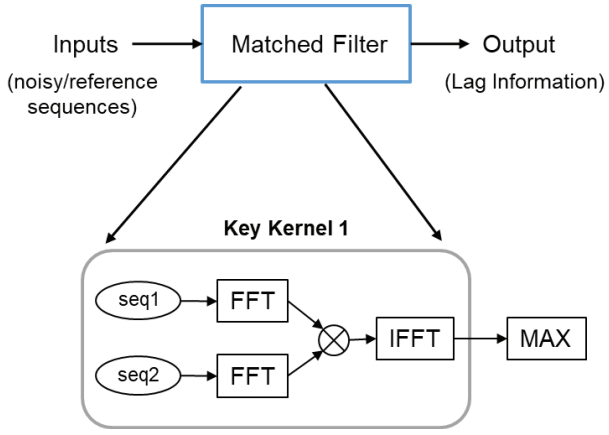


Figure 17: Radar Correlator Flowchart

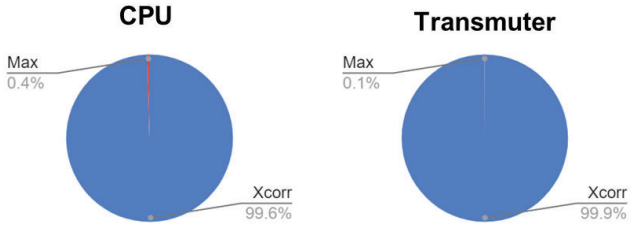


Figure 18: Radar Correlator Profiling Results

Table 4: Radar Correlator Simulation Results

	Energy Efficiency (GFLOPS/W)	Execution Time (ms)
CPU	0.03	1.39
GPU	0.07	0.18
TM at 1.0 GHz	2.92	0.38
TM at 0.2 GHz	4.70	1.50

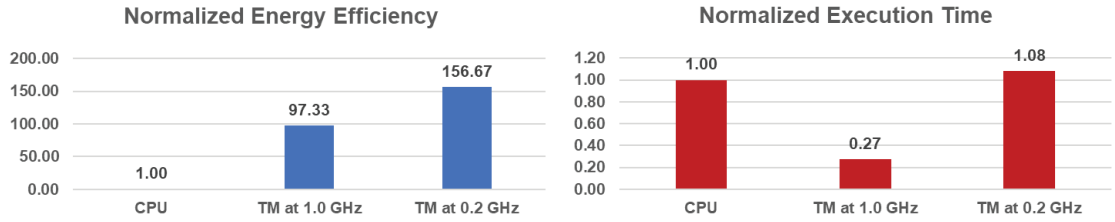


Figure 19: Normalized Simulation Results for Radar Correlator

Correctness. We evaluate the MSE between outputs of TM implementation and the reference C code in DSSoC DASH (ASU) repository. The MSE is 0.0000.

3.5 Pulse Doppler

Pulse Doppler Radar uses pulse-timing techniques to determine the range of a target. It uses the Doppler effect of the returned signal to determine the target object's velocity. It has wide applications in surveillance and healthcare. The algorithm runs for m iterations, corresponding to the number of pulses. In each iteration, a signal of length 2 * *n* is processed where *n* is the length of each pulse. Its main computation kernels are FFT-256 and IFFT-256, for which we use our radix-4 FFT implementation.

Flowchart and profiling results. The flowchart of Pulse Doppler Radar is shown in Figure 20. It takes *m* noisy signal inputs and a reference signal of length *n* and performs matched filter for a total of *m* times to get a 2D time sequence of size *m x n*. Then, it performs FFT of size *m* across the different sequences to create the Doppler map. The target information corresponds to the maximum value. Figure 21 shows that the *MatchedFilter* process dominates the time cost.

Simulation results. We implement Pulse Doppler on TM and compare it with the CPU reference C implementation both in terms of energy efficiency (GFLOPS/W) and execution time (ms). The detailed results are shown in Table 5. The normalized comparison results are provided in Figure 22. Results show that TM running at 1.0 GHz takes 82% less execution time while achieving 66.8x higher energy efficiency compared to the CPU baseline. TM running at 0.2 GHz takes approximately half time and achieves 207x higher energy efficiency compared to the CPU baseline.

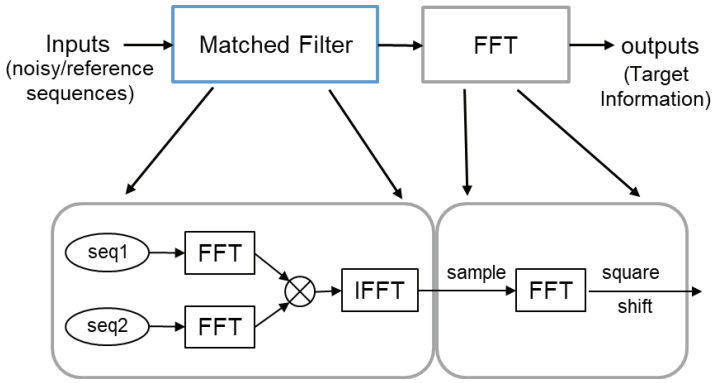


Figure 20: Pulse Doppler Radar Flowchart

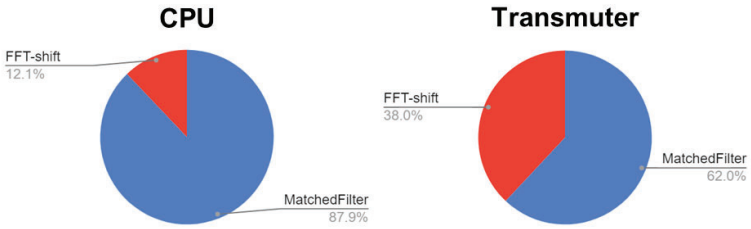


Figure 21: Pulse Doppler Radar Profiling Results

Table 5: Pulse Doppler Radar Simulation Results

	Energy Efficiency (GFLOPS/W)	Execution Time (ms)
CPU	0.04	81.6
TM at 1.0 GHz	2.67	15.0
TM at 0.2 GHz	8.28	46.9

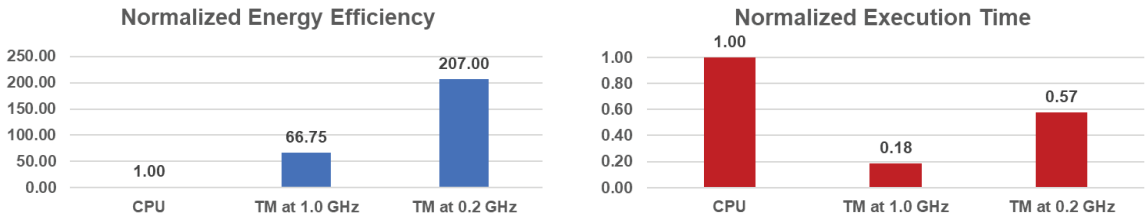


Figure 22: Normalized Simulation Results for Pulse Doppler Radar

Correctness. We evaluate the MSE between outputs of TM implementation and the reference C code in DSSOC-DASH (ASU) repository. The MSE is 0.0009.

3.6 SAR range Doppler Algorithm

Synthetic-Aperture Radar (SAR) is a coherent phase array radar system that is usually placed in an airborne moving platform. It takes advantage of the motion and coherent nature of the system to simulate a much larger aperture than the actual aperture size. Range Doppler Algorithm (RDA) does coherent processing over multiple waveforms to detect the range and azimuth of the target between each pulse. It stacks the radar return of each pulse to create a 2D matrix. A Range Doppler map is created by passing a matched filter across time and taking azimuthal FFT along the pulses. The algorithm consists of three computation blocks: Range compression, Azimuth FFT, and Azimuth Compression. Its computation kernels are FFT-256, IFFT-256, FFT-512, and IFFT-512. We use our radix-4 FFT implementation for FFT/IFFT-256 and radix-2 FFT implementation for FFT/IFFT-512.

Flowchart and profiling results. The flowchart of SAR RDA is shown in Figure 23. The 2D matrix obtained by stacking the radar returns of 256 pulses is of size 256 x 512. The first step is range compression, where matched filter is done with reference signal A. Each matched filter computation consists of both FFT-512 and IFFT-512. The resulting 2D array is then passed through the azimuthal FFT block, where FFT-256 is computed along the pulses. Finally, Azimuth compression multiplies the shifted frequency signal with reference signal B and converts it

back to time domain by making 512 calls to IFFT-256. Figure 24 shows that all stages share a comparable portion of time cost.

Simulation results. We implement SAR RDA on TM and compare it with the CPU reference C implementation both in terms of energy efficiency (GFLOPS/W) and execution time (ms). The detailed results are shown in Table 6. The normalized results are provided in Figure 25. Results show that TM running at 1.0 GHz takes 33% less execution time while achieving 11.5x higher energy efficiency. TM running at 0.2 GHz takes approximately 2.67x more time but achieves 33.4x higher energy efficiency compared to the CPU baseline.

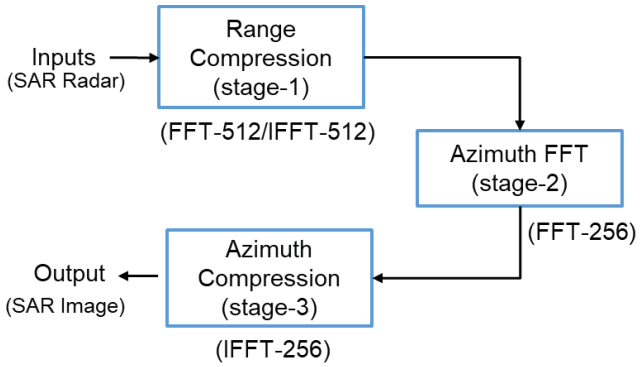


Figure 23: SAR Range Doppler Algorithm Flowchart

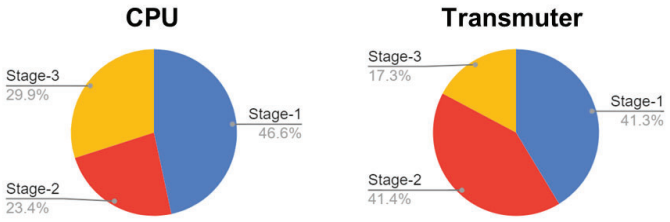


Figure 24: SAR RDA Profiling Results

Table 6: SAR RDA Simulation Results

	Energy Efficiency (GFLOPS/W)	Execution Time (ms)
CPU	0.14	62.2
TM at 1.0 GHz	1.61	41.0
TM at 0.2 GHz	4.67	155.6

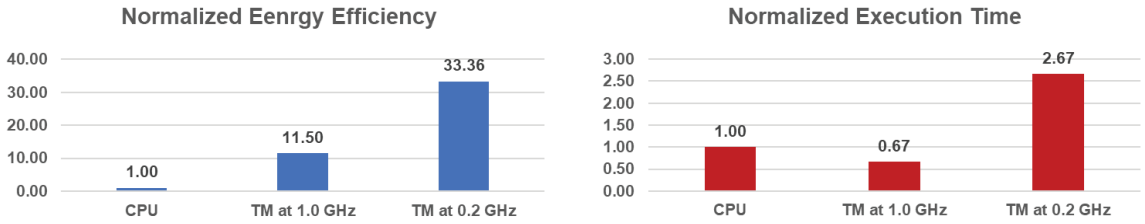


Figure 25: Normalized Simulation Results for SAR RDA

3.7 SAR Backprojection

SAR Backprojection is used for all image geometries and does not require a separate motion compensation step. Compared to the original three-stage implementation (IFFT, phase correction, linear interpolation), the TM implementation uses precomputed phase correction parameters to save computation at the expense of higher storage. Note that the phase correction vector depends on the current position of the radar and changes but not at the frame-level granularity. Our implementation of SAR Backprojection consists of two computation blocks: Inverse FFT and Linear Interpolation. The main computation kernel is IFFT-512, for which we use our radix-2 FFT implementation.

Flowchart and profiling results. The flowchart of SAR Backprojection is shown in Figure 26. The SAR inputs are processed by the IFFT module followed by phase correction (where we use precomputed parameters), and then finally linear interpolation to construct the SAR image as the output. As shown in Figure 27, we find that IFFT dominates the CPU time while linear interpolation dominates the TM time. We hypothesize that the linear interpolation involves irregular memory accesses on a large data structure, which is not favorable for TM.

Simulation results. We implement SAR Backprojection on TM and compare it with the CPU reference C implementation both in terms of energy efficiency (GFLOPS/W) and execution time (ms). The detailed results are shown in Table 7. The normalized comparison results are provided in Figure 28. Results show that TM running at 1.0 GHz takes 40% less execution time while achieving 11.8x higher energy efficiency. TM running at 0.2 GHz takes approximately 2.80x more time but achieves 27.4x higher energy efficiency compared to the CPU baseline.

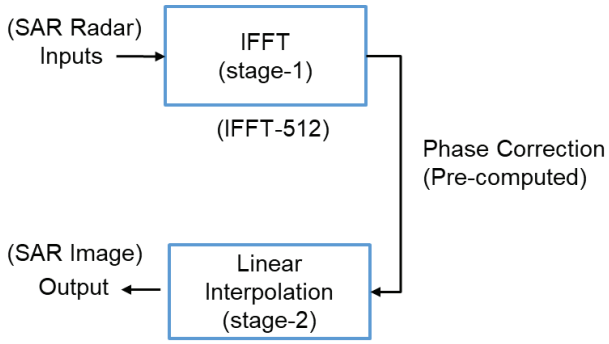


Figure 26: SAR Backprojection Flowchart

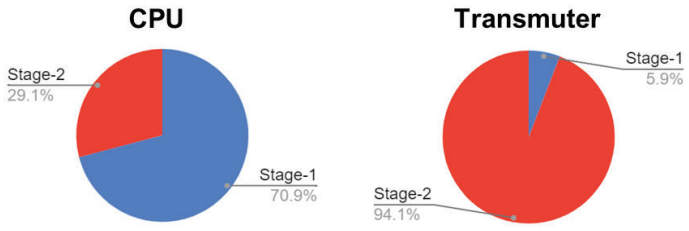


Figure 27: SAR Backprojection Profiling Results

Table 7: SAR Backprojection Simulation Results

	Energy Efficiency (GFLOPS/W)	Execution Time (ms)
CPU	0.49	95.0
TM at 1.0 GHz	5.80	58.2
TM at 0.2 GHz	13.4	280.4

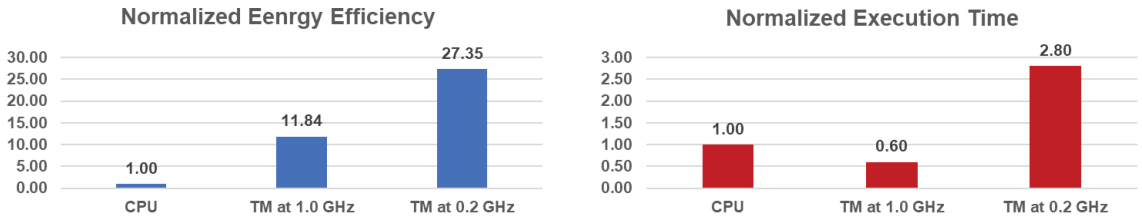


Figure 28: Normalized Simulation Results for SAR Backprojection

3.8 HIVE benchmark: VGG 19

VGG19 is a convolutional neural network (CNN) proposed for large-scale image recognition. The purpose of this microbenchmark is to stress test data access and matrix computations for a relatively large CNN model. The benchmark includes three layers taken from the VGG-19 model that uses the largest sized matrix multiplications, as shown in Figure 29. It includes the following

- Two 2D convolution layers conv1_2 and conv2_1.
- One max-pooling layer between two conv2D layers.

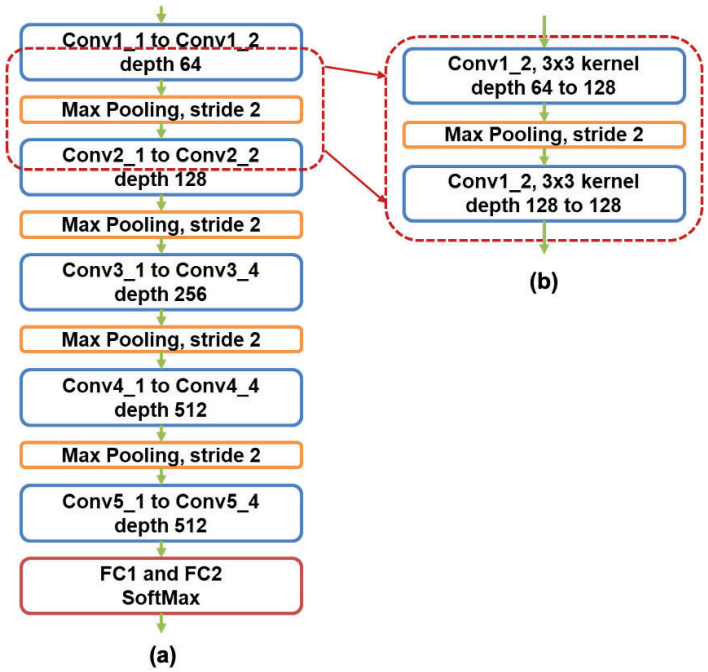


Figure 29: (a) VGG 19 Structure [A.1], (b) HIVE VGG 19 Benchmark

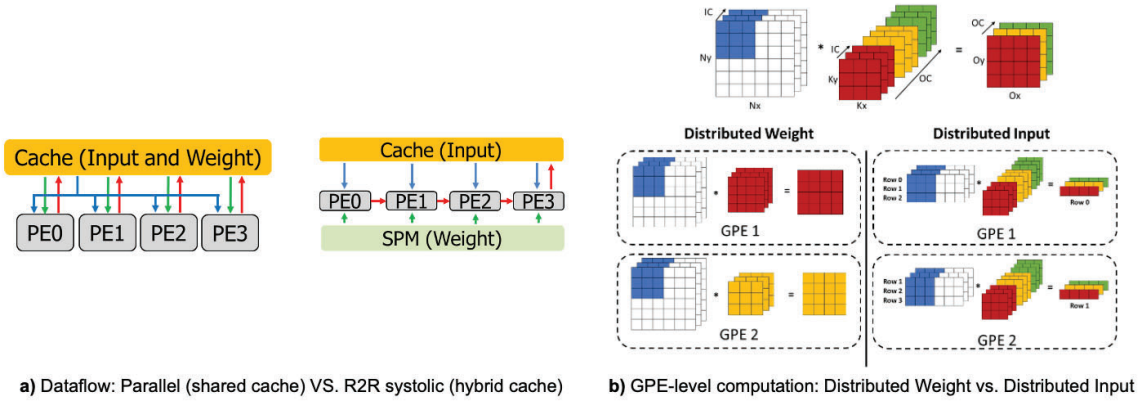
The tensor sizes and layer parameters for batch size of N are as follows:

- The input tensor is [N, 64, 224, 224].
- Conv1_2 uses 3x3 kernels, stride of 1, and padding of 1. The layer has 64 input channels and 128 output channels.
- Max pooling layer uses kernel size of 2, stride of 2.
- The input tensor size of Conv2_1 is [N, 128, 112, 112].
- Conv2_1 uses 3x3 kernels, stride of 1, and padding of 1. The layer has 128 input channels and 128 output channels.
- The output tensor is [N, 128, 112, 112].

3.8.1 Transmuter implementation with reconfiguration

To accelerate the 2D convolution layer with different parameters, the following reconfiguration options are considered:

- Cache modes: Private cache, shared cache, hybrid cache (shared + scratchpad)
- Dataflow: Parallel dataflow or R2R systolic dataflow, as is shown in Figure 30a.
- Tile-level mapping and GPE-level computation: distributed weight and distributed input, as is shown in Figure 30b.



a) Dataflow: Parallel (shared cache) VS. R2R systolic (hybrid cache) **b)** GPE-level computation: Distributed Weight vs. Distributed Input

Figure 30: Reconfiguration Options. a) Dataflow, b) GPE-level Computation

2D convolution (Conv2D) layers in CNN models have a large diversity in the sizes of the inputs and weights -- they differ from network to network and across layers in a network. The shape of the input and weight tensors affects Transmuter performance; performance differs from one configuration to another. With run-time reconfiguration, Transmuter can operate using the optimal configuration for all layers, as is shown in Figure 31a. To determine the optimal configuration at runtime, a decision-tree-based engine is used to predict the optimal configuration [A.2]. As is shown in Figure 31b, it uses the parameters of the 2D convolution layer as input and outputs the configuration options that achieve the optimal performance.

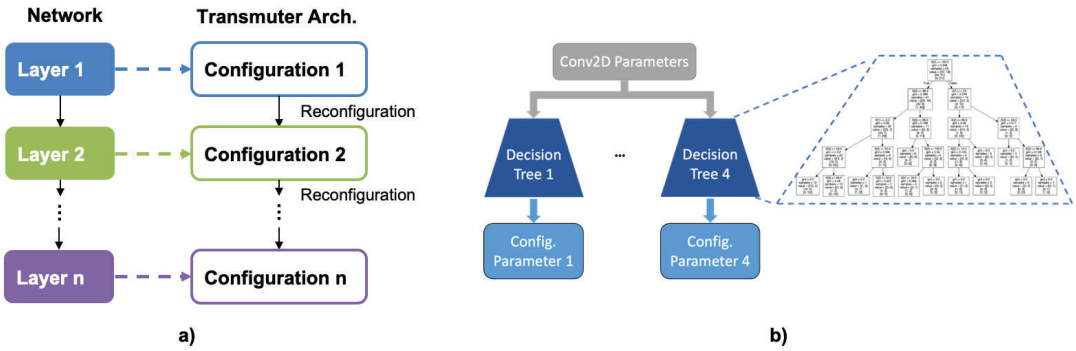


Figure 31: a) Runtime Reconfiguration Between Layers. b) Decision Tree Configuration Predictor [A.2]

In this evaluation, we use energy efficiency (GFLOPS/W) as the performance metric. For this benchmark, the predicted optimal implementation for the baseline Transmuter configuration (4 tiles and 16 GPEs per tiles, L1 and L2 cache bank of size 4 kB) for both conv1_2 and conv2_1 uses shared cache mode for L1 and L2, parallel dataflow, distributed weight on tile level, and distributed input on GPE computation.

3.8.2 Performance Evaluation

CPU configuration. Intel Core i9-10850K (Intel 14nm technology); Core frequency 5.0 GHz; implemented using PyTorch.
GPU configuration. NVIDIA RTX 8000 (TSMC 12nm technology); Core frequency 1.8 GHz; implemented using PyTorch with CUDA.
Transmuter (TM) configuration. 4 tiles with 16 GPEs per tile. 4KB L1 and L2 cache banks. GPE operating frequency is optimized to achieve the optimal energy efficiency (GFLOPS/W): Convolution layers run at 0.2 GHz and max-pooling runs at 1.0 GHz.
The performance of VGG 19 TM implementation is shown in Figure 32, where a) shows the energy consumption comparison and b) shows the energy efficiency comparison. At the optimal frequency for energy efficiency, Transmuter achieves energy efficiency of 29.65 GFLOPS/W with 0.19J energy consumption. The energy efficiency is 26.5x compared to CPU and is 7.7x compared to GPU.

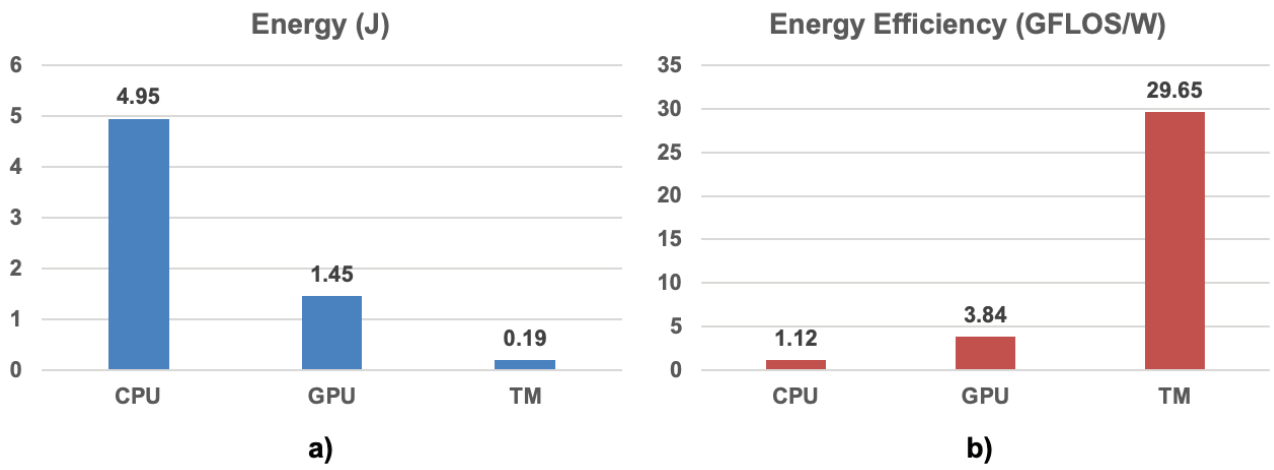


Figure 32: Performance Evaluation of VGG19. Comparison with CPU and GPU with respect

(a) Total Energy Consumption and (b) Energy Efficiency

Performance comparison with CPU and GPU if Transmuter clock frequency is set at 1 GHz.

At 1GHz, the Transmuter execution time is 0.65s, energy is 0.48J and energy efficiency is 11.55 GFLOPS/W. Thus compared to CPU, its execution time is 35.6x higher and its energy efficiency is 10.3x higher. Compared to GPU, its execution time is 118.6x higher and its energy efficiency is 3.0x higher.

References

A.1. K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," International Conference on Learning Representations, 2015.

A.2. Y. Xiong, J. Li, D. Blaauw, H.-S. Kim, T. Mudge, R. Dreslinski, and C. Chakrabarti. Improving Energy Efficiency of Convolutional Neural Networks on Multi-core Architectures through Run-time Reconfiguration. IEEE International Symposium on Circuits and Systems (ISCAS), 2022.

3.8.3 Introduction

ResNet is a class of convolutional neural networks (CNN) that solve the gradient vanishing problem for deep CNNs. It uses residual blocks with input bypass to prevent the gradient from vanishing. Common ResNet model includes ResNet18, ResNet34, ResNet50, etc., where the number indicates the number of 2D convolution (conv2D) layers in the model. In this evaluation, we use ResNet18 as a showcase to show the performance of ResNet on Transmuter.

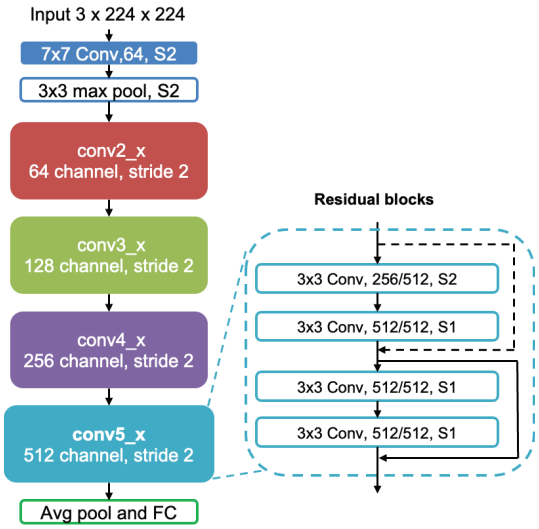


Figure 33: ResNet18 Structure [B1]

ResNet18, as is shown in Figure 33, uses 18 2D convolution layers. The 2D convolution layers are grouped into 4 residual blocks, each block includes multiple residual bypasses. For ImageNet, the input size is 3x224x224. The input size of the different 2D convolution layers vary from 224x224 to 7x7. Three different convolution kernel sizes are used: 7x7, 3x3 and 1x1. The input/output channel numbers vary from layer to layer, from 3 channels to 512 channels.

3.8.4 Parameterized ResNet: Implementation and Performance

The implementation of ResNet 18 is optimized with run-time reconfiguration. The optimal configuration of each 2D convolution layer is predicted by the decision-tree-based decision engine [B.2]. The following configurations are used for ResNet 18 implementaiton:

- Shared L1, parallel, distribute input/weight: Conv2D layer 1
- Hybrid L1, parallel, distribute weight /weight: 1x1 Conv2D layers
- Hybrid L1, R2R, distribute input/weight: 3x3 Conv2D layers in block 2 and 3
- Hybrid L1, R2R, distribute weight/input: 3x3 Conv2D layers in block 4 and 5

3.8.5 Performance Evaluation

CPU configuration. Intel Core i9-10850K (Intel 14nm technology); Core frequency 5.0 GHz; implemented using PyTorch.
GPU configuration. NVIDIA RTX 8000 (TSMC 12nm technology); Core frequency 1.8 GHz; implemented using PyTorch with CUDA.
Transmuter configuration. 4 tiles with 16 GPEs per tile. 4KB L1 and L2 cache banks. GPE operating frequency is optimized to achieve the optimal energy efficiency (GFLOPS/W): Convolution layers run at 0.2 GHz and average pooling and fully connected layers run at 1.0 GHz.

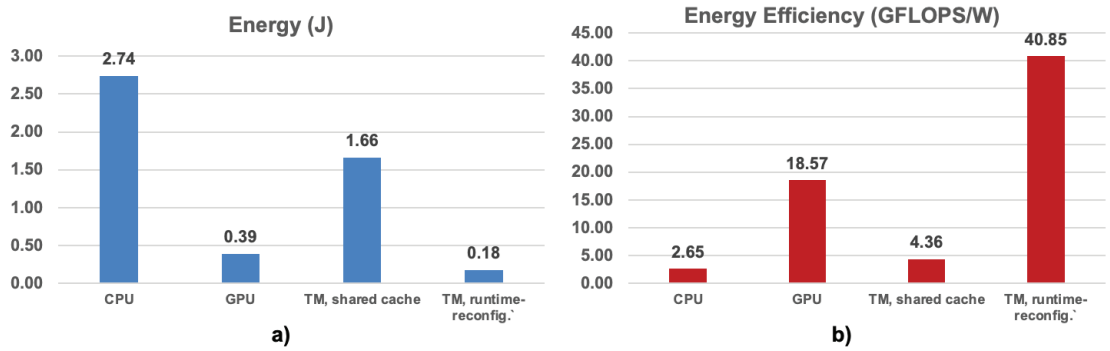


Figure 34: Performance Evaluation of ResNet18

Comparison with CPU and GPU with respect to (a) total energy consumption and (b) energy efficiency.

The performance of ResNet 18 TM implementation is shown in Figure 34, where a) shows the energy consumption comparison and b) shows the energy efficiency comparison. With run time reconfiguration, TM achieves energy efficiency of 40.85 GFLOPS/W which is 9.4x compared to TM shared cache implementation without reconfiguration. The energy efficiency is 15.4x compared to CPU and is 2.2x compared to GPU.

Performance comparison with CPU and GPU if Transmuter clock frequency is set at 1 GHz.

At 1GHz, the Transmuter execution time is 0.64s, energy is 0.48J and energy efficiency is 15.13 GFLOPS/W. Thus compared to CPU, its execution time is 50.7x higher and its energy efficiency is 5.7x higher. Compared to GPU, its execution time is 339x higher and its energy efficiency is 1.23x lower.

References

B.1. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778

B.2. Y. Xiong, J. Li, D. Blaauw, H.-S. Kim, T. Mudge, R. Dreslinski, and C. Chakrabarti. Improving Energy Efficiency of Convolutional Neural Networks on Multi-core Architectures through Run-time Reconfiguration. IEEE International Symposium on Circuits and Systems (ISCAS), 2022.

3.8.6 Introduction

DefConv refers to 2D deformable convolution, which is a powerful component in high-accuracy object detection systems. It uses trained offsets to achieve adaptive receptive convolution fields to efficiently extract the features in the input feature maps as is shown in Figure 35a. Compared to standard 2D convolution, here data access pattern is data-dependent and sparse, making it more challenging. The two steps of DefConv are as follows:

- A stand-alone 2D convolution is applied to the input to generate the offset field.
  - The convolution kernel is applied to the shifted input, where the shift is a function of the offset.

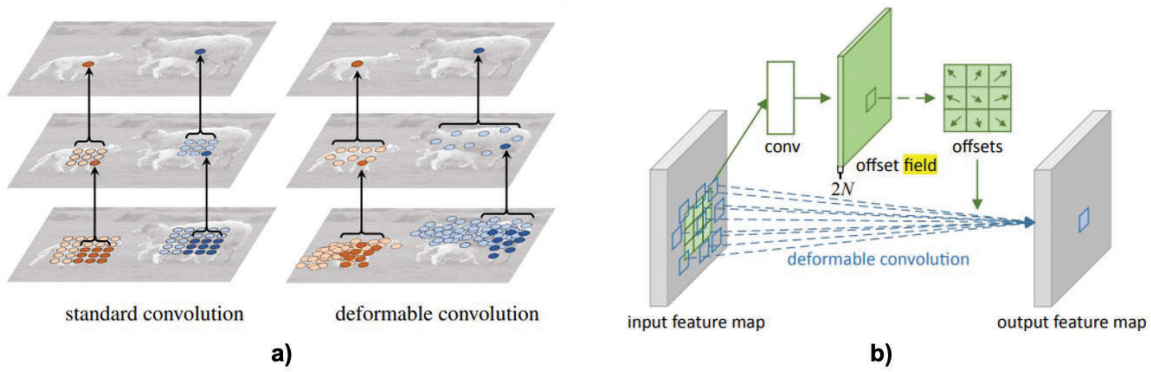


Figure 35: a) Receptive area: Standard Convolution vs. Deformable Convolution, b) A 3x3 Deformable Convolution [C.1]

3.8.7 Transmuter Implementation

Deformable convolution is implemented on Transmuter using shared cache 2D convolution implementation. The L1 and L2 cache are configured as shared cache. The implementation uses parallel dataflow where each tile/GPE works independently. The tile level uses distributed weight and GPE-level uses distributed input. The offset values are pre-computed and stored in the main memory, and during computation, the offset values are loaded in the shared cache.
In this evaluation, the benchmark layer parameters are as follows:

- Input size (h, w): 64×64, 128×128 and 256×256
- Kernel size (k): 3×3
- Input/output channel (In_channel/Out_channel): 16, 64 and 256
- Offset range: -5 to 5, -10 to 10, -20 to 20

3.8.8 Performance Evaluation

CPU configuration. Intel Core i9-10850K (Intel 14nm technology); Core frequency 5.0 GHz; implemented using NumPy.

Transmuter configuration. 4 tiles with 16 GPEs per tile. 4KB L1 and L2 cache banks. GPE operating frequency is 0.2 GHz chosen to achieve the optimal energy efficiency (GFLOPS/W).
Note that we do not provide GPU evaluation since the NumPy code provided does not support GPU execution.

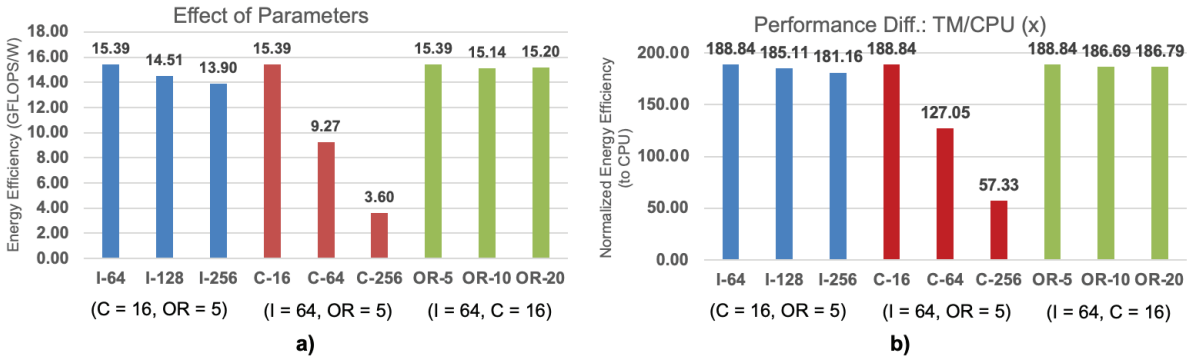


Figure 36: Defconv Performance. a) Effect of Different Parameters, b) Normalized Enregy Efficiency Compared to CPU

Figure 36 shows the performance of TM DefConv implementation. Figure 36a shows the effect of each parameter. It shows that as the input size increases, the energy efficiency is slightly decreased. The channel number has a large effect on the energy efficiency. As the channel number increases, the energy efficiency is significantly decreased. Offset range (OR) has little effect on energy efficiency, the performance under OR = 5, 10 and 20 are very close.

Figure 36b shows the normalized energy efficiency of TM compared to CPU. The CPU energy efficiency does not change much under different parameters (0.06~0.08 GFLOPS/W). Under different parameter combinations, TM achieves 57.33x to 188.84x energy efficiency compared to CPU. Overall, TM outperforms CPU more when the input size and channel numbers are small. However, as the parameter sizes increase, the performance difference is not as much.

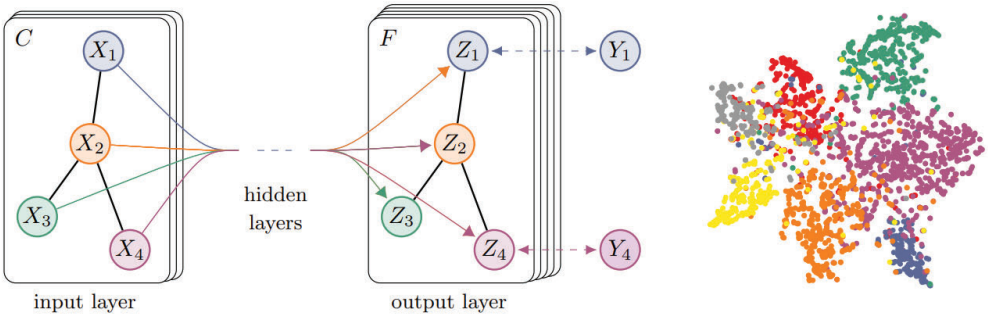
Performance comparison with CPU if Transmuter clock frequency is set at 1 GHz.
At 1GHz, the Transmuter execution time varies from 0.6x to 2.4x compared to CPU under different parameter combinations. For C=16, I=64 and OR=5, the execution time is 0.0063s, energy is 0.0058J and energy efficiency is 6.47 GFLOPS/W. Thus compared to CPU, its execution time is 16.3x lower, and its energy efficiency is 80.8 x higher.

References

C.1. Jifeng Dai, et al. "Deformable convolutional networks." Proceedings of the IEEE Int. Conf. on Computer Vision, 2017.

3.8.9 Introduction

Graph neural network (GNN) is a powerful tool for extracting useful features from graph-structured data, such as social networks. In this report, we focus on implementing the most popular GNN - graph convolutional network (GCN) [D.1] on Transmuter. We then compare the performance of Transmuter implementation with the CPU/GPU baseline implementation for the SDH-Prog-Eval-2021 benchmark.



(a) Graph Convolutional Network (b) Hidden layer activations

Figure 37: GCN is used in extracting useful features from graph-structured data [D.1]

3.8.10 Transmuter Implementation

GCN takes graph-structured data as input and feeds it to a series of GCN layers. The processing of each layer consists of a combination step (Dense matrix-matrix multiplication or DMM) and an aggregation step (Graph operation). A Two-layer GCN can be represented as follows [D.1]:

$$Z = f(X, A) = \text{softmax}\Big(\hat{A}\ \text{ReLU}\Big(\hat{A}XW^{(0)}\Big)\ W^{(1)}\Big)$$

For a graph G(V, E), X denotes the feature matrix, where the row index represents nodes and each row represents the feature vector of the corresponding node; A represents the adjacency matrix of G that has a dimension of |V| by |V|; $W^{(0)}$ and $W^{(1)}$ are the learnable weight matrices of the GCN.

The combination step of $XW^{(0)}$ is done by a single DMM kernel. It has regular data access and is straightforward. The aggregation step involves graph traTransmuter v1.0l with irregular data access. It consists of visiting the neighbor of each node, aggregating the features of neighboring nodes and saving the new feature vector of that node. Since the adjacency matrix is large, we use pull-mode graph traTransmuter v1.0l implementation. We assume that the data is in compressed storage column (CSC) format.

We present three different mapping modes for aggregation. They are (a) node parallelism, where different GPEs work on the traTransmuter v1.0l of different nodes, (b) feature parallelism, where GPEs work collectively to aggregate features, and (c) hybrid parallelism, where the LCP in a tile traverses the neighbors of a node and the GPEs in a tile aggregate the features of different nodes. Figure 38 describes the three modes.

For small feature size, node parallelism is most suitable, as different GPEs work on separate nodes. In this mode, each GPE performs a pull operation on node V, where the pull operation includes simply adding up the feature vectors of all nodes that point to node V and the feature vector of node V itself. However, we notice that if the feature vector size is very large (i.e. over 1,000), node parallelism will result in a large cache miss rate. For such cases, feature parallelism is more suitable. In this mode all GPEs work on the pull operation of a single node V, and different GPEs work on addition operations on different sections of the feature vector. Hybrid

parallelism is a combination of node parallelism and feature parallelism, where node parallelism is applied across tiles, and feature parallelism among GPEs in a tile.

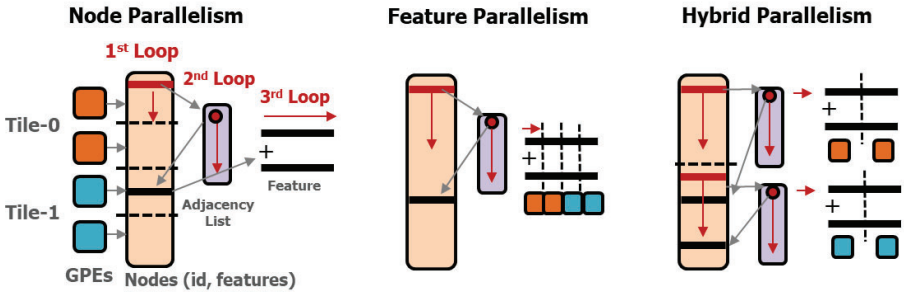


Figure 38: Three Different Implementations for the Aggregation Step

3.8.11 Performance Evaluation

For evaluation, we use the public dataset Cora, Slashdot-0811, and Rmat18 datasets. As shown in Table 8, we set the feature size and the neuron size (second dimension of weight matrices) to 32, and set the number of layers to 2. This is according to the default value settings in the code provided in the SDH-prof-eval-2021 benchmark.

Table 8: Dataset & GCN Settings

Dataset	Node	Edge	Layer-1		Layer-2	
			N1	F1	F2	N2
Cora	2708	10556	32	32	32	32
Slashdot-0811	77360	469180	32	32	32	32
Rmat18	174147	3800348	32	32	32	32

For the DMM kernel implementation, we follow typical GEMM mapping. We do not use loop-tiling because feature size is small and there is no performance benefit due to tiling. For the aggregation phase, we use node parallelism for all datasets. As shown in Figure 39, we sweep four different cache modes and find that with L1-cache in shared mode and L2-cache in private mode (SP setting) achieves the smallest time cost for both aggregation and combination phases. So we present the layer-by-layer results of the Cora dataset using the SP mode in Table 9.

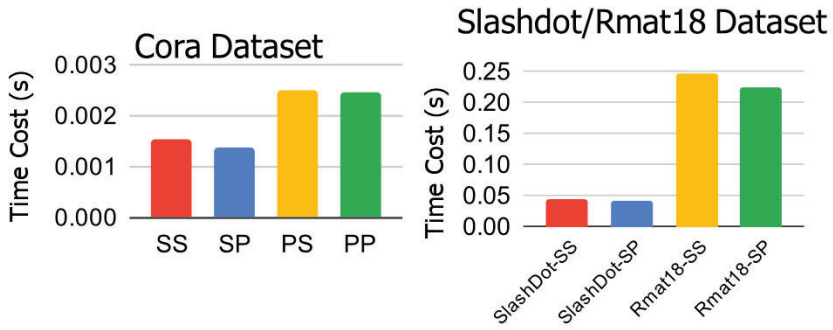


Figure 39: Transmuter Time Cost Under Different Cache Modes

Table 9: Layer Level Performance Breakdown for Cora Dataset

Cora Dataset (SP mode)		Time	Power	GFLOPS/W	GOPS/W
Layer-1	Stage1-DMM	0.0004726	0.741	15.83	57.94
	Stage2-Graph	0.0002248	0.722	2.08	21.19
Layer-2	Stage1-DMM	0.0004613	0.741	15.87	58.06
	Stage2-Graph	0.0002237	0.722	2.09	21.28

From Table 9, we find that the execution patterns for the two layers are almost identical. More importantly, the first stage DMM is much more energy-efficient than the graph traTransmuter v1.0l kernel for the Cora dataset. We also evaluate the time that is spent in each of the stages for all three datasets. Figure 40 presents the execution time results. From this figure we see that Stage 2 is dominant for dense graphs, as in Rmat18.

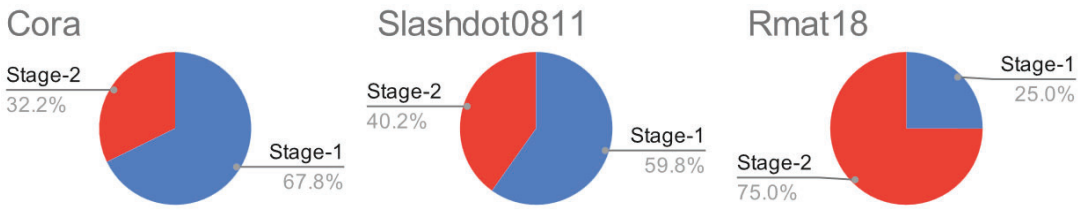


Figure 40: Time Cost Proportion of the Two Stages for All Three Datasets

Performance comparison with CPU and GPU.
CPU configuration. AMD Ryzen 7 5800X (7nm node) running at 3.8GHz; C++ implementation
GPU configuration. NVIDIA RTX 3090 (7nm node) running at 1.7GHz; CUDA implementation.
Transmuter configuration. 4 tiles with 16 GPEs per tile. 14nm technology node. Clock frequency is 1 GHz. Both L1 and L2 have cache bank size of 4KB. We choose the L1-L2 cache configuration with the best performance. In this case, it is L1 in shared mode and L2 in private mode.

Table 10: Comparison between TM and CPU/GPU (1.0 GHz)

Workload		Time (s)	Power (W)	GFLOPS/W	GOPS/W	Energy (J)
Cora	CPU	0.00042	63	0.441	1.781	0.02646
	GPU	0.00096	116	0.104	0.4208	0.11136
	TM	0.00138	0.735	11.5	46.28	0.0010143
SlashDot0811	CPU	0.0763	86	0.0529	0.2198	6.5618
	GPU	0.0227	184	0.0829	0.3446	4.1768
	TM	0.0408	0.739	11.5	47.88	0.0301512
Rmat18	CPU	0.4771	97	0.0207	0.1046	46.2787
	GPU	0.1595	183	0.0328	0.1659	29.1885
	TM	0.2251	0.720	5.90	29.88	0.162072

The results for all three datasets are shown in Table 10. Here, we assume the operation and FLOP numbers for all three hardware are the same since they perform an equivalent amount of work. From the results, we can see Transmuter achieves GFLOPS/W energy efficiency from 26x ~ 285X compared to CPU and 110x~180x compared to GPU.

Next, we sweep the clock frequency from 100MHz to 1GHz and find that Transmuter has the best performance at 200MHz. Table 11 presents the results. The GFLOPS/W energy efficiency ranges from 60.6x~763x compared to CPU and 256x ~ 481x compared to GPU.

Table 11: Comparison between TM and CPU/GPU (0.2 GHz)

Workload		Time (s)	Power (W)	GFLOPS/W	GOPS/W	Energy (J)
Cora	CPU	0.00042	63	0.441	1.781	0.02646
	GPU	0.00096	116	0.104	0.4208	0.11136
	TM	0.0057	0.077	26.7	107.5	0.0004389
SlashDot0811	CPU	0.0763	86	0.0529	0.2198	6.5618
	GPU	0.0227	184	0.0829	0.3446	4.1768
	TM	0.161	0.077	27.9	115.6	0.012397
Rmat18	CPU	0.4771	97	0.0207	0.1046	46.2787
	GPU	0.1595	183	0.0328	0.1659	29.1885
	TM	0.790	0.076	15.8	79.21	0.06004

References

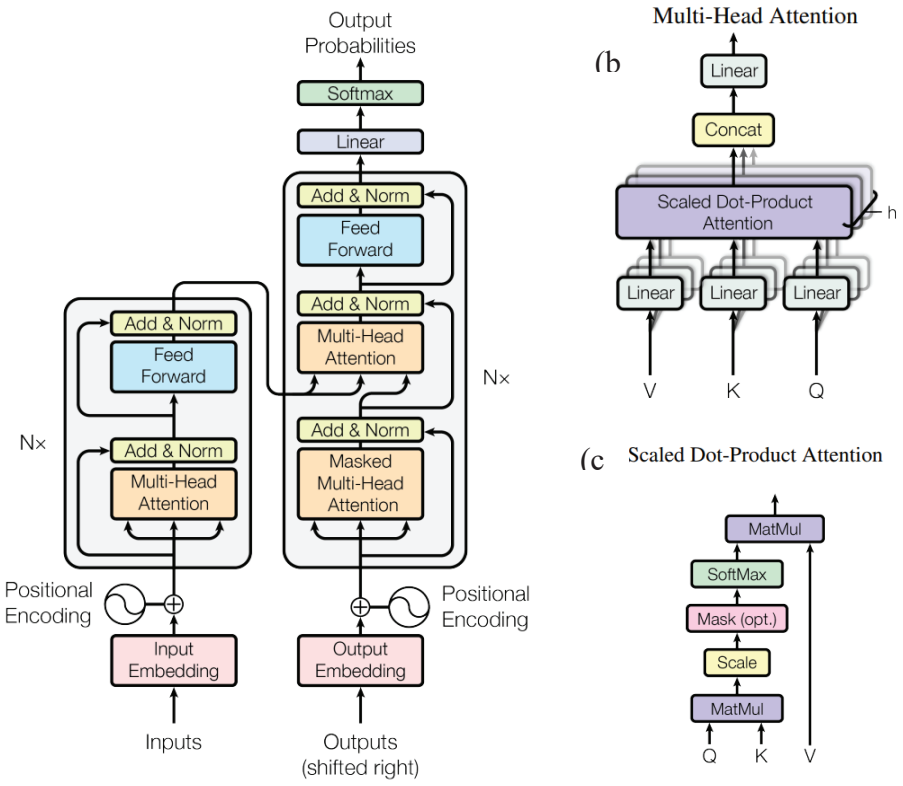
D.1 Thomas N. Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks." Int. Conf. on Learning Representations (ICLR) 2017.

3.8.12 Background

Transformer is a simple network architecture based solely on attention mechanisms, omitting recurrence and convolutions entirely [E.1]. Compared to the state-of-the-art, Transformer shows better machine translation quality while being more parallelizable and requiring significantly less time to train. Transformer architecture consists of two parts: encoder and decoder, as shown in Figure 41a:

- Encoder is composed of N=6 encoder layer (Figure 41a left), where each encoder layer has two sub-layers. The first sub-layer is a Multi-Head Attention layer, and the second is a fully connected feed-forward layer. Residue connections are used between the two sub-layers.
- Decoder is also composed of N=6 decoder layers. A decoder layer is similar to an encoder layer, but with an extra Masked Multi-Head Attention. The Masked Multi-Head Attention ensures the predition of position *i* only uses output from positions less than *i*.
- A Multi-Head Attention layer (Figure 41b) takes three matrices V, K and Q, performs linear projection on each of them, and feeds them to a Scaled Dot-Product Attention layer (Figure

41c), where matrix multiplication, scaling and softmax are performed. The above oprations are repeated multiple times (thus called multi-head), and the resuls are concatenated before feeding to the last linear projection layer.



(a) Transformer model architec-

Figure 41: a) The Transformer Model Architecture, b) Details of Multi-Head Attention layer and c) Details of Scaled Dot-Product Attention [E.1]

3.8.13 Benchmark Network

The Transformer network used in CPU and GPU baseline consists of N=6 transformer layers, each layer has 1) one Multi-Head Attention layer, with number of attention heads being 16 and number of attention hidden dimension being 64, and 2) one feed-forward layer with two linear layers. The first layer changes the embedding dimension from 1024 to hidden dimension 2048, and the second layer changes from 2048 to 1024. The input data has 32 batches, each batch is of size 32, the image size is 256, and patch size is 32. The network size and input graph sizes are chosen to be large enough to reveal the true performance of GPU.

For Transmuter (TM) simulation, the tasks are distributed to GPEs at the granularity of mini batch. To finish simulation in reasonable time, we shrink the number of batches from 32 to 2, and keep the same batch size at 32. We also shrink feed-forward inner layer dimension from 2048 to 32 (scaled by 64), attention head number from 16 to 4 (scaled by 4), and attention hidden

dimension from 64 to 4 (scaled by 16). This ensures that the attention layer and feed-forward layer are shrunk with the same ratio of 64. The input data is scaled by 16, model depth is scaled by 6 and model size is scaled by 64, resulting an in overall scaling factor of 6144.

3.8.14 Performance Evaluation

Figure 42 shows the time breakdown for each kernel in the Transformer network. The execution time of Transformer is dominated by linear layers, which are essentially GEMM operations, and so overall Transformer performance is dominated by the GEMM performance. We simulate TM with two configurations: 1) parallel data flow with both L1 and L2 in shared cache mode, and 2) systolic array mode dataflow with L1 acting as buffers between PEs, and L2 in shared cache mode. Configuration 1 has good performance since shared cache mode has good data reuse from sharing weights among all GPEs. The weaker performance of systolic array mode is because a lot of time is spent fetching data to be passed through the GPEs one-by-one. The R2R systolic array mode in TM has less data fetch overhead and will be investigated next.

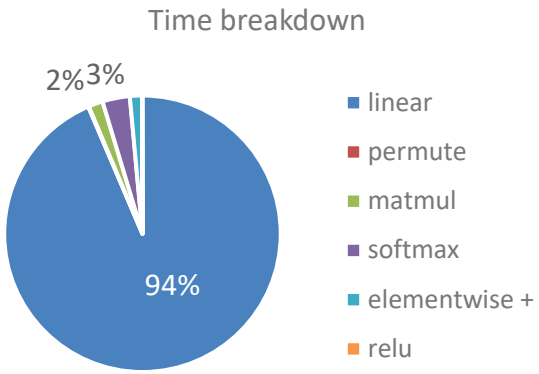


Figure 42: Transmuter Simulation Time Breakdown for Each Kernel

Performance comparison with CPU and GPU.
CPU configuration. Intel i7-7700 (14nm node); CPU running at 4.2 GHz.
GPU configuration. Nvidia Tesla V100 GPU (12 nm node); running at 1.245 GHz.
Transmuter configuration. 4 tiles with 16 GPEs per tile. Both L1 and L2 use shared cache mode, where the cache bank size is 4KB. The clock frequency is set at 0.2 GHz. We scale the execution time of Transmuter by 6144, the ratio of model size and input data size used for Transmuter and CPU/GPU, for a fair comparison.

Figure 43 shows that the CPU baseline has an energy efficiency of 5.77 GFLOPS/W, and GPU has an energy efficiency of 40.03 GFLOPS/W. Transmuter achieves an energy efficiency of 38.01 GFLOPS/W, which is 6.6x higher than that of CPU and 1.05x lower than that of GPU. The superior performance of GPU comes as no surprise since the main kernel for Transformer is GEMM which is well-optimized for GPU.

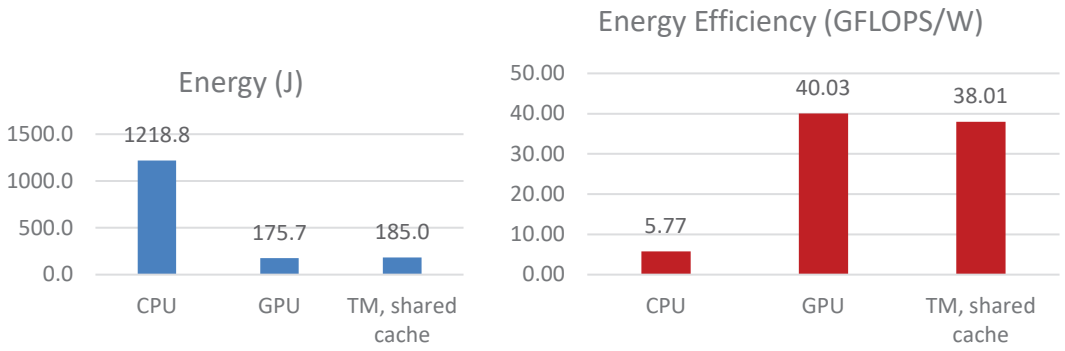


Figure 43: Performance Evaluation of Transformer
Comparison with CPU and GPU with respect to (a) total energy consumption and (b) energy efficiency.

Performance comparison with CPU and GPU if Transmuter clock frequency is set at 1 GHz.
At 1GHz, the Transmuter execution time is 949s, energy is 690J and energy efficiency is 10.19 GFLOPS/W.
Thus compared to CPU, its execution time is 50.6x higher and its energy efficiency is 1.766x higher.
Compared to GPU, its execution time is 1621.3x higher and its energy efficiency is 3.92x lower.

References
E.1 Ashish, Vaswani, et al. "Attention Is All You Need" *Advances in Neural Information Processing Systems 30*. 2017.

3.9 Sparse matrix-matrix Multiplications

There are three key approaches to matrix-matrix multiplication – inner-product, outer-product and row-wise multiplication.

Inner-Product Approach: The traditional approach to matrix-matrix multiplication is the inner-product multiplication. In inner-product approach, each row of matrix A performs a vector multiplication with each column of matrix B to produce each element of the result matrix C, as shown in Figure 44. However, this method can be highly inefficient when computing with sparse matrices, as the nonzero elements of each row and columns needs to have matching index in order to be multiplied together to generate the partial products.

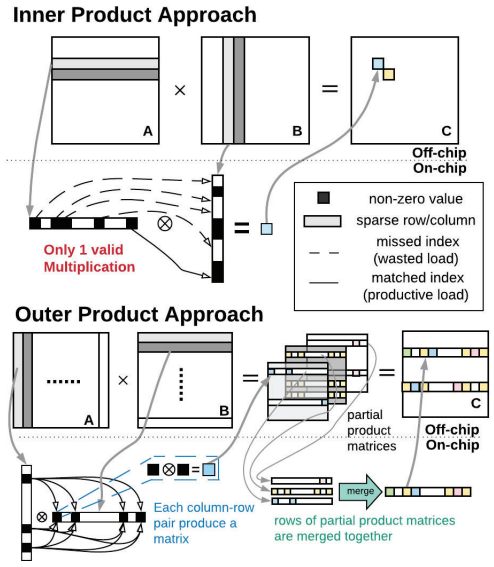


Figure 44: Overview of Inner-Product and Outer-Product Multiplication

Outer-Product Approach: In the outer-product method, each column of matrix A is multiplied with corresponding row of matrix B to generate a partial product matrix of the result matrix C, as shown in Figure 44. Once all the partial-product matrices are computed, they are all merged together to produce the final result matrix. With the outer-product method, the nonzero elements of each input matrix only need to be accessed once.

Row-wise Multiplication Approach: In the row-wise sparse matrix-matrix multiplication, each row of the matrix A is multiplied with the row of matrix B that corresponds to the index of the nonzero elements of the given row, resulting in a row of the result matrix C. Given a row of matrix A, each nonzero element in the row accesses the row of matrix C based its column index and perform a scalar vector multiply to produce the partial-product row. All the partial-product rows of a row of matrix A accumulate into a single row of matrix C corresponding to the same row position of the row of matrix A. Since the rows of matrix B are fetched again for each row of matrix A and the nonzero elements of matrix A can have overlapping column indices, it is quite likely for rows of matrix B to be accessed multiple times throughout the computation. Figure 45 shows an overview of row-wise multiplication scheme.

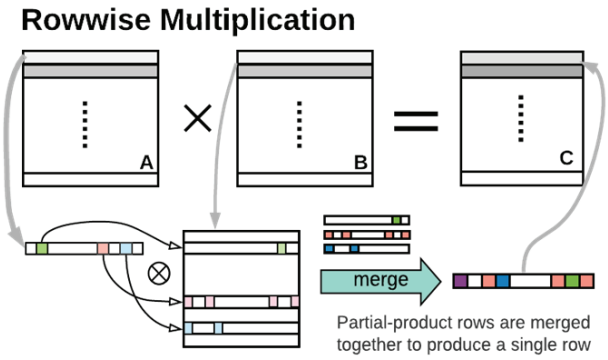


Figure 45: Overview of Row-wise Multiplication

3.9.1 Evaluation of Sparse Matrix-Matrix Multiplication on GCN-type datasets

We consider sparse matrix-matrix multiplications for the case when the two multiplicands have distinct structures, as in the case of Graph Convolutional Networks (GCN). The first (left) multiplicand is a sparse or ultra-sparse adjacency matrix following power-law, and the second (right) multiplicand is a sparse or dense matrix that contains node features.

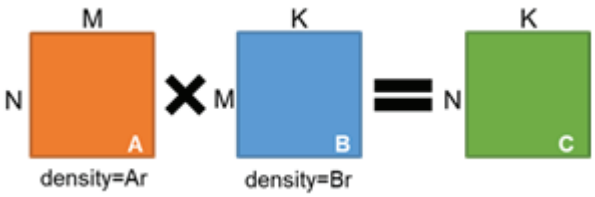


Figure 46: SpMM Workload where the Multiplicand Matrices have Different Structures

We evaluate the performance of inner-product, outer-product, and row-wise multiplications. Table 12 describes the Transmuter configuration and the different matrix configurations. We choose the matrix A to be the Citeseer dataset, which has size 3327x3327 with density 0.08%. The matrix B is derived by varying K from 64 to 1024 (feature dimensions) and density of 1% to 100% (feature densities).

Table 12: Overview of TM Configuration and Matrix Configurations for SpMM

Workload Description	Feature Reduction in CiteSeer N=M=3327, Ar=0.08%, K = (64, 256, 1024), Br = (1%, 10%, 100%)
Transmuter Configuration	4x16 tile architecture, 4KB L1 - 4KB L2 Shared or private cache modes
Best Performing Cache Configuration	Inner product (private - private) Outer product (shared - private) Row-wise product (private - private)

Figure 47 describes the time performance of the three approaches. In each approach, the best performing cache configuration listed in Table 12 is chosen. The figure shows that row-wise

multiplication outperforms outer-product in every case. This is due to the smaller memory footprint in the merge phase of row-wise multiplication compared to the outer-product based method.

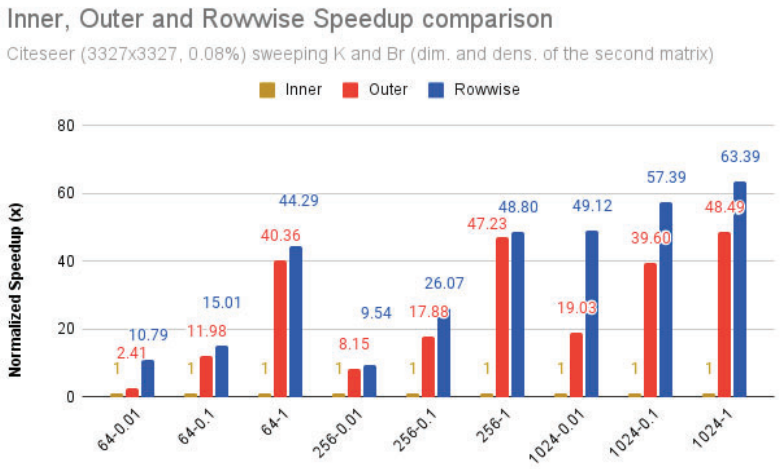


Figure 47: Speedup Comparison of Outer-product and Row-wise Multiplication Normalized to Inner Product for Different Values of K (64, 256, 1024) and Br (1%, 10%, 100%) for Citeseer Dataset

Next we studied row-wise and outer-product based methods in the SpMM implementation for larger GCN. We operated on a larger adjacency matrix of size 19717x19717 and 0.022% density corresponding to the Pubmed dataset. The K dimension of matrix B is varied from 64 to 1024 (feature dimensions) and density Br from 1% to 100% (feature densities). The speedup normalized to outer-product dataflow is shown in Figure 48.

As discussed earlier, the row-wise dataflow only differs from the outer-product dataflow in the merge phase. It incurs a smaller memory footprint since it accumulates 1 row at a time in 1 GPE (or 1 GPE group if R2R is enabled). As the workload of 1 row increases (the dimension and density of 1 row of feature elements increase), the difference between the row-wise multiplication scheme and the outer-product scheme reduces. Specifically, the speedup of row-wise over outer-product drops from 3.3x to 1.2x as we sweep (K-Br) from 64-0.01 to 1024-1. The reason is that even though row-wise dataflow has a smaller memory footprint in the merge phase, the growing working set (with the increase of the dimension and density of 1 row) can still be larger than the cache size and the performance will diminish to the outer-product where we will accumulate the whole matrix size. But the row-wise dataflow opens other optimization opportunities such as load balancing and reordering when applying the merge, since the merge phase is broken into granularity of rows.

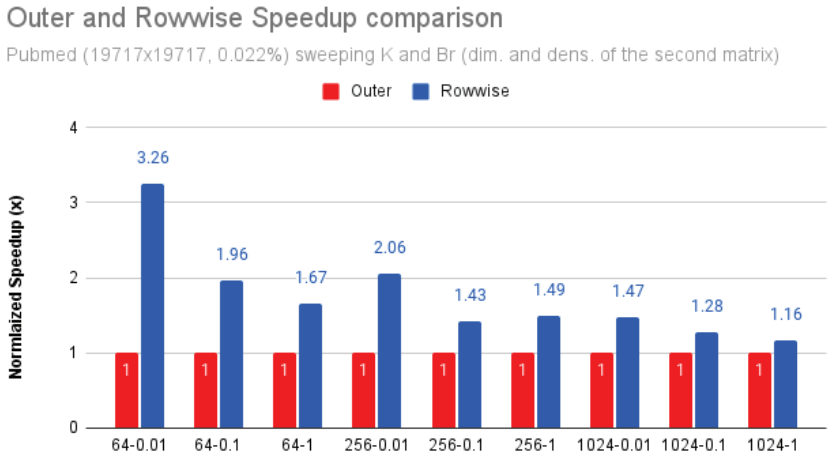


Figure 48: Speedup Comparison Row-wise Method Normalized to Outer Product Method for Pubmed Dataset with N=M=19717, Ar=0.022%, K=64, 256, 1024 and Br=1%, 10% and 100%

Next we apply DVFS for row-wise multiplication on Pubmed dataset for the case when N=M=19717, Ar=0.022%, K=1024, Br=10%. The Transmuter configuration is 4 tiles with 16 GPEs per tile; L1 and L2 cache banks operate in private-private mode and are each of size 4KB. We see that the GOPS/W increases from 47.16 GOPS/W when operated at 1 GHz to 127.51 when operated at 0.1 GHz – an improvement of 2.71x. The energy efficiency also increases from 0.18 GFLOPS/W when operating at 1GHz to 0.5 GFLOPS/W when operating at 0.1 GHz, saturating at around 0.2 GHz. Thus use of DVFS help improve energy efficiency of row-wise SpMM implementations.

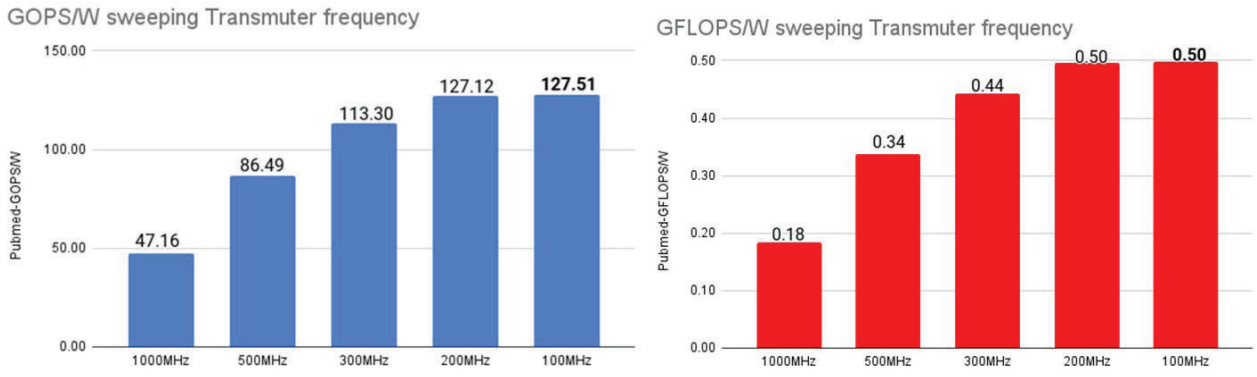


Figure 49: Energy Efficiency as a Function of Frequency when Row-wise Based SpMM is Implemented for Pubmed Dataset with N=M=19717, Ar=0.022%, K=1024, Br=10%

3.9.2 Sparse Matrix-Matrx Multiplication using Row-wise Multiplication – a Deep Dive

In the row-wise multiplication of SpMM, the rows of matrix A are distributed evenly across the tiles of the Transmuter, and each tile splits those rows amongst their GPEs. Each GPE fetches the nonzero elements of the rows of matrix A, and the corresponding rows of matrix B. While the nonzero elements of matrix A are fetched only once, the rows of matrix B can be fetched multiple times throughout the computation. Each GPE performs the row-wise multiplication to generate the partial-product rows, then performs the merge to produce the corresponding row of matrix C. The final row is written to memory, and the GPE proceeds to the next row. Figure 50 describes the mapping scheme.

When distributing the rows across the GPEs, the work distribution can be **fixed** or **dynamic.** In fixed distribution, each row of matrix C is assigned a fixed number of processing units, and the workload is always distributed evenly among the same set of processing units. In dynamic distribution, rows are assigned to GPEs on-the-fly as they are free to process more work, allowing for better load balancing throughout the execution.

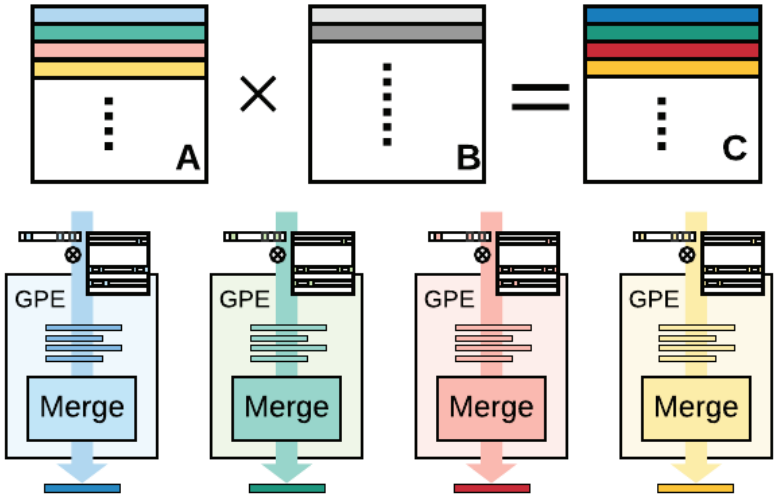


Figure 50: Mapping of Row-wise Multiplication on Transmuter

Merge Algorithms: Row-wise method has three variations for the merge portion of the algorithm: dense vector, sorting list, and systolic. In dense vector merge the partial results are written onto a dense vector array that can be indexed directly. After the multiply computation is finished, the dense vector is converted to sparse format and written to memory. Sorting list merge is the same algorithm as the merge stage of the outer-product method, where the heads of the partial products are pushed onto a sorting list to keep extracting the smallest element. Systolic merge is a more distributed version of the sorting list merge, where the merge operation is spread across multiple GPEs. Each GPE push its smallest element onto the next GPE through the R2R queue.

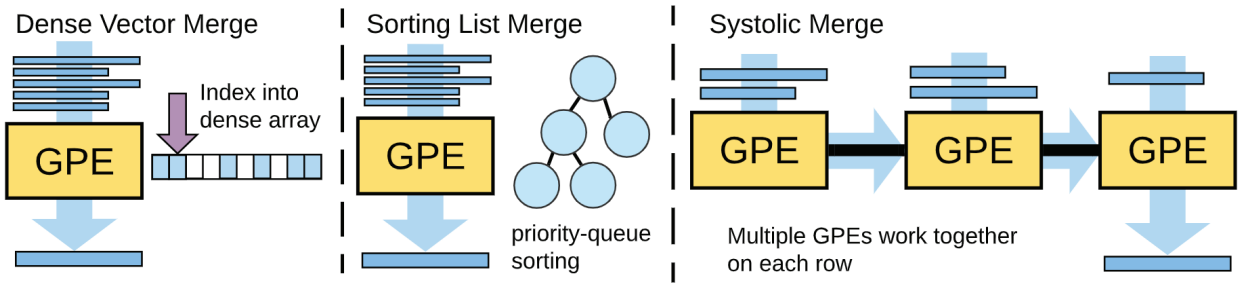


Figure 51: Merge Algorithm Options for Row-wise Multiplication

Results for Sparse Matrix-Matrix Multiplication.

The performance of outer-product method and the row-wise approach is compared for N=M=K=1000, 5000 and 10,000, and density r varying from 0.001 to 0.005 in Figure 52. Overall, row-wise multiplication exhibits up to 7.17x better performance over the outer-product method across all the matrix size and densities. Comparing the different merge algorithms of the row-wise approach, the dense vector merge performs better for denser and larger matrices, while the sorting list merge is the most optimal for smaller and sparser matrices.

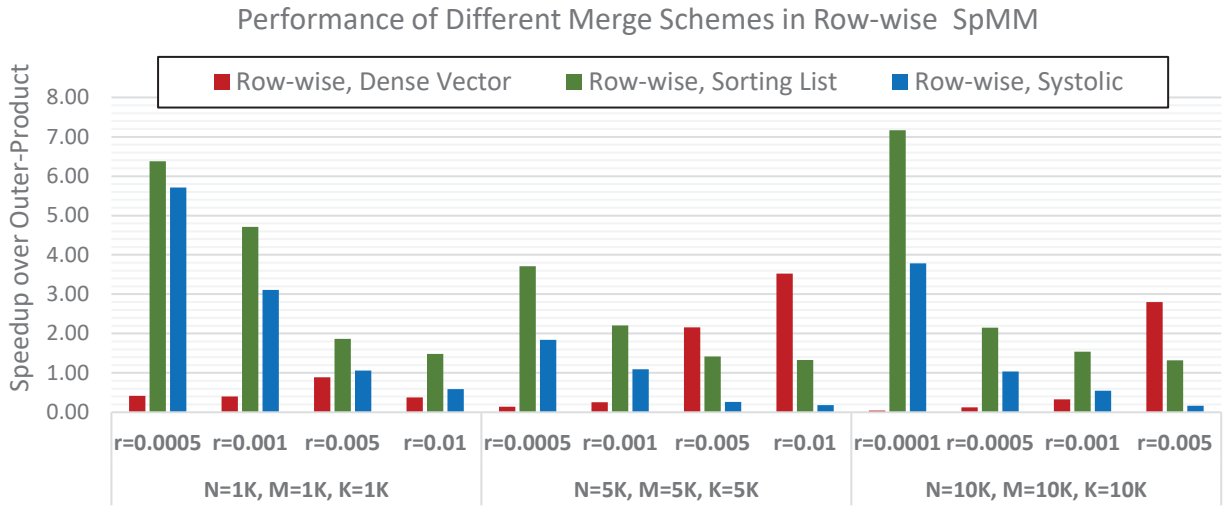


Figure 52: Performance of Different Merge Algorithms of Row-wise Multiplication

Results for Graph Convolutional Networks.

Graph Convolutional Networks (GCNs) are an important deep learning workload for analyzing graph data. The forward propagation of multi-layer spectral GCN comes in the form: $X_{l+1} = \sigma(AX_lW_l)$, where A is the adjacency matrix of the target graph, X_l is the input feature matrix for layer-l, W_l is the weight matrix of layer-l, and σ is the non-linear activation function. The adjacency matrix A is highly sparse with density less than 0.3%. The feature matrix X_l also starts with high sparsity of greater than 90% in its first layer, but becomes progressively denser at deeper layers due to weight matrix W being a fully dense matrix. Thus, AX_l is a sparse matrix-matrix multiplication with a highly sparse adjacency matrix A and a rectangular feature matrix X with a narrow width and increasing density.

The AX_l portion of GCN kernel was tested by taking the CORA citation dataset (2708-by-2708 graph of academic publications and their citations) as the adjacency matrix A as shown in Figure 53. For the feature matrix X, a 2708-by-64 matrix was generated with density varying from 10% to 100% to reflect the increasing density of the feature matrix at deeper layers of the GCN. Row-wise approach with sorting list merge exhibits the best performance for densities less than 30%, but systolic merge starts to outperform the sorting list for the feature matrix density higher than 30%. This is due to the higher density of the feature matrix leading to more merging of partial products that share the same column index, which gets distributed across the systolic pipeline for parallel execution. Dense vector merge exhibits a flat execution time across all densities, and starts outperforming the systolic merge for feature matrices denser than 65%.

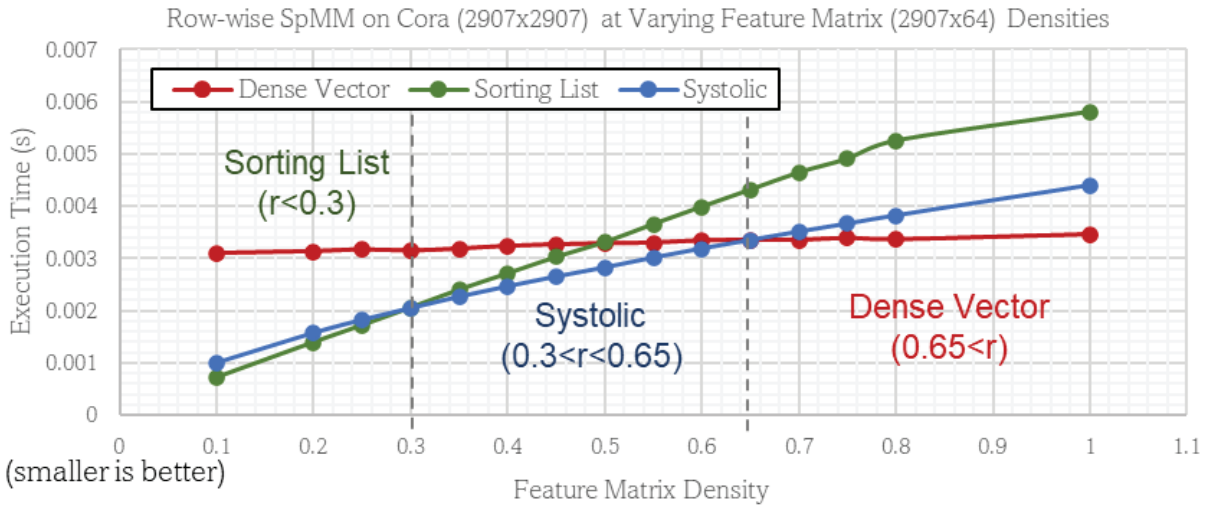


Figure 53: Performance of Row-wise Multiplication on GCN Workload

Matrix Re-ordering

Matrix reordering algorithms reconstruct sparse matrices by altering the organization of its non-zero elements to aid in future processing or visual representation. The revere Cuthill-McKee algorithm (RCM) produces a matrix will less bandwidth by concentrating the nonzero elements towards the diagonal. The rabbit ordering also reduces the bandwidth of the matrix, but also orders the nonzero elements into more number of smaller local clusters. The results of applying the RCM and rabbit reordering algorithms on the Cora citation database matrix is shown in Figure 54. The original Cora matrix has most of its nonzero elements evenly spread out, while the RCM algorithm concentrates the nonzero elements in small section along the diagonal. While the rabbit ordering still has high concentration of nonzero elements in the center, there are still some local clusters on the other regions of the matrix.

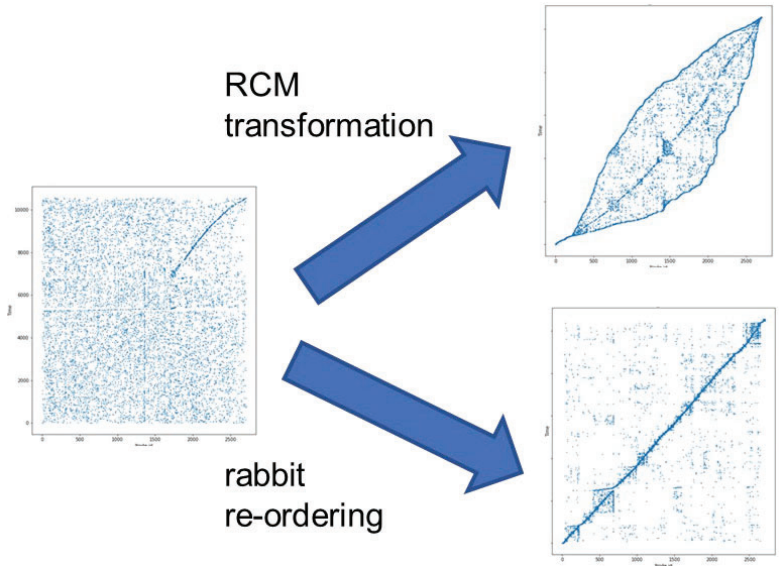


Figure 54: Matrix Re-ordering on Cora Citation Matrix

The result of applying the rabbit matrix reordering on the Cora adjacency matrix is shown in Figure 55. For both dense vector and sorting list merge of row-wise multiplication, the matrix reordering had no noticeable impact on the performance of the algorithm. However, systolic merge showed significant improvement in performance, where the overall performance of the algorithm improved by 11-14%. Matrix reordering favors systolic merge because systolic merge can take advantage of the local clusters by splitting the work in those denser regions among multiple GPEs to have better load balancing. Overall, the rabbit order expands the region in which systolic merge performs the best from 30-65% to 20-70%.

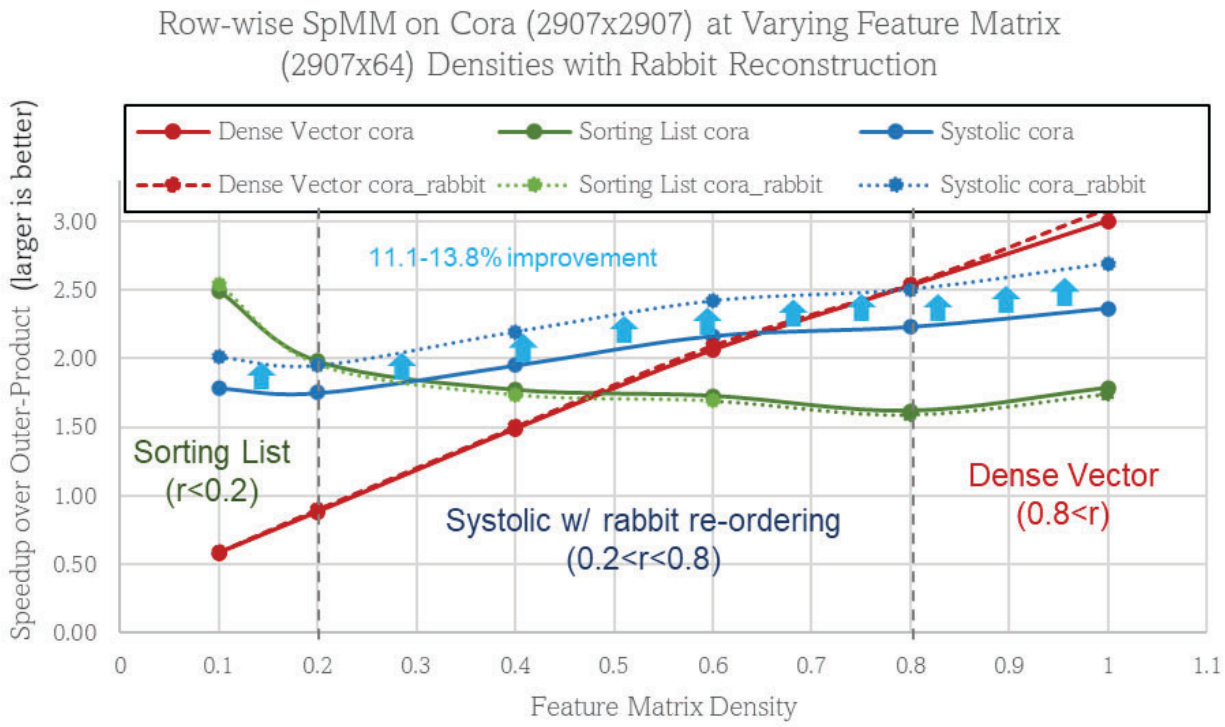


Figure 55: Performance of Row-wise Multiplication on GCN Workload with Matrix Re-ordering

3.10 Collaborative filtering (CF)

Collaborative filtering (CF) is a widely used algorithm in recommendation systems. The relation between users and objects are modeled as a weighted bipartite graph. CF algorithm computes a score for each user and object. The score is updated iteratively by taking into account the previous score of all neighboring nodes and the weights of corresponding edges (between current node and neighboring nodes)

The CF algorithm has the best performance in pull mode, where each node pulls the information from neighboring nodes and aggregates them locally to update the score. Operating in pull mode requires the graph to be presented in compressed sparse column (CSC) format. The mapping assigns vertices evenly to each GPE across 4 tiles (all 64 GPEs). Since all vertices are activate at each iteration, all traTransmuter v1.0ls are meaningful and therefore, no wasting operations.

Performance of the CF algorithm improves with use of the Prodigy-based prefetcher designed at U Mich. First, there is no skipping operation in pull mode, which avoids prefetched data being wasted if a traTransmuter v1.0l is skipped. Second, CF works on weighted graphs where edge weights have to be fetched as well as so the prefetcher can be used to get more relevant data.

Results: The graph dataset consists of weighted random uniform graphs with 10K and 20K vertices and average out-degree of 5, 10 and 20. The Transmuter configuration is 4x16 (4 tiles with

16 GPEs per tile) with L1 banks of size 4 KB in shared cache mode and L2 banks of size 4KB L2 in shared cache mode.

We ran the CF implementation with and without prefetcher and reported the execution time and energy efficiency in Figure 56 and Figure 57, respectively. We see that the prefetcher helps improve the performance of small graphs (10K nodes). The energy efficiency performance degrades for large graphs (20K nodes for both systems – with and without prefetcher, with the prefetcher version having worse performance. This is because a large graph stresses the L1 cache and use of a prefetcher stresses the L1 cache even more.

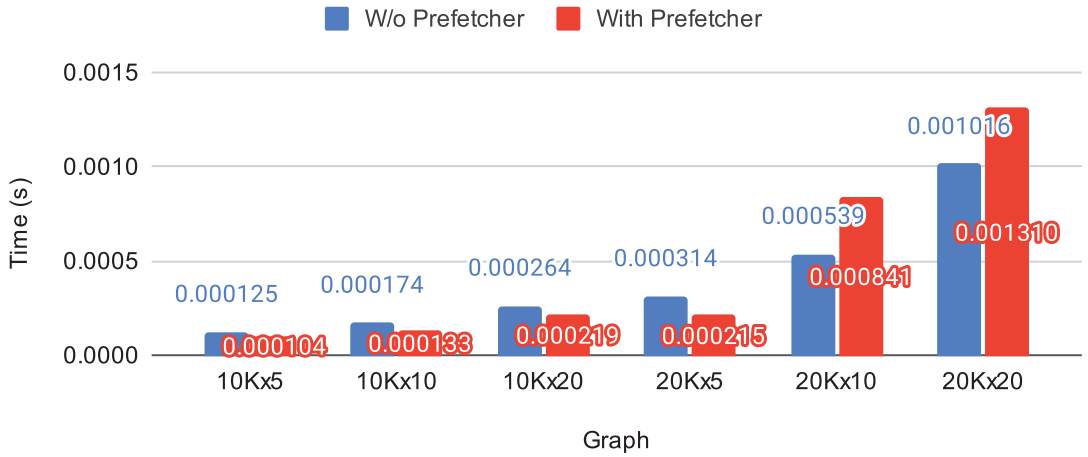


Figure 56: Execution time of Collaborative Filtering with/without Prefetcher

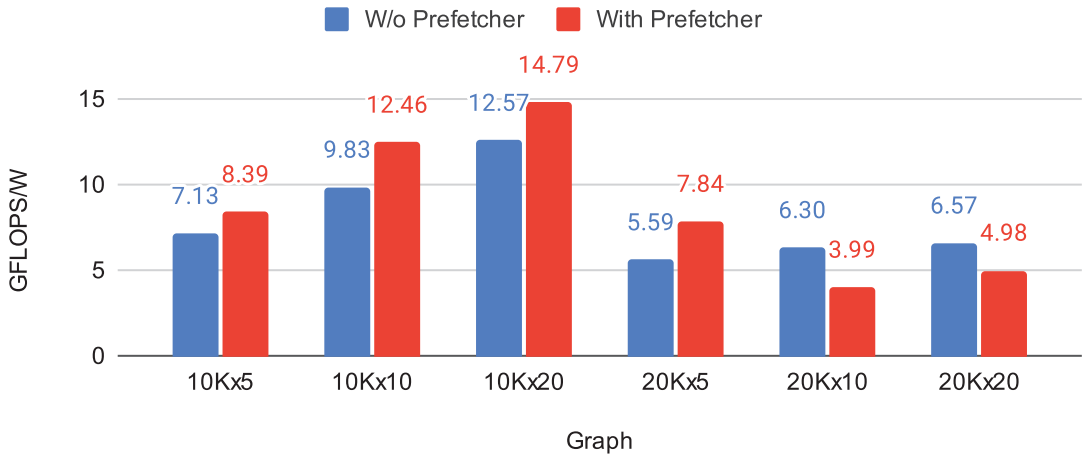


Figure 57: Energy Efficiency of Collaborative Filtering with/without Prefetcher

3.10.1 Exploiting Reconfiguration Opportunities: Dynamic Graphs

The graph dataset in real world applications, especially graphs that model social network and communication networks, can be dynamic. Such graphs may change in sparsity, which might

result in sub-optimal configuration if the implementation is kept static. Thus, implementation might need to adapt to the changing input data.

We investigated this case for two graph workloads, Single Source Shortest Path (SSSP) and aggregation phase of Graph Convolution Network (GCN). We simulated the dynamic dataset by using a set of static graphs with different sparsities and evaluated their performance on Transmuter.

Example Single Source Shortest Path (SSSP) implemented with quadratic BFS. A toy example is shown in Figure 58, where we use the level array to mark the frontier nodes and the distance array to store the shortest distance. SSSP can operate in two modes: push and pull.

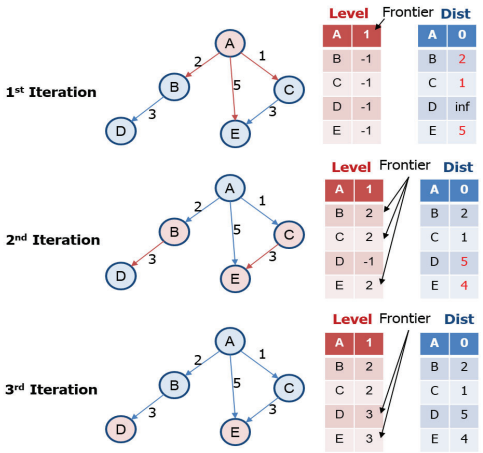


Figure 58: A Toy Example of Single Source Shortest Path

Pull mode SSSP consists of three steps: (1) Check whether a node is a frontier based on level array. (2) Visit the neighboring nodes of the frontier nodes. (3) Check Distance (frontier current -> neighbor node), update if needed. (update if the current distance is shorter).

Pull Mode SSSP consists of three steps: (1) Visit the neighboring nodes of the current node. (2) Check whether a neighboring node is a frontier. (3) Check Distance (frontier neighbor -> current node), update if needed. The choice of whether to use pull mode or push mode depends on the number of edges as will be demonstrated.

Results: The static graph datasets to simulate a dynamic graph for SSSP consists of 5 weighted graphs with 10K nodes and varying out-degree of 2, 4, 8, 16 and 32. The Transmuter configuration is 4x16 (4 tiles with 16 GPEs per tile) with L1 banks of size 4 KB in shared cache mode and L2 banks of size 4KB L2 in shared cache mode.

We compare the performance of the following configurations: Baseline-Pull, Baseline-Push, Adapt mode which chooses between Baseline-Pull and Baseline-Push modes. We see that for small out-degree, the pull mode is better but for large out-degree, the push mode is better. The optimal mode shifts from pull mode to push mode when the outdegree is 8 for this dataset. In addition, we increase the L1 cache to 8 KB to further improve the Adapt mode performance. The results are shown in Figure 59. In the chosen dataset, the point at which the optimal mode

switches from pull mode to push mode occurs around 8 out-degree. Overall, the Adapt mode shows a great advantage in reducing total execution time.

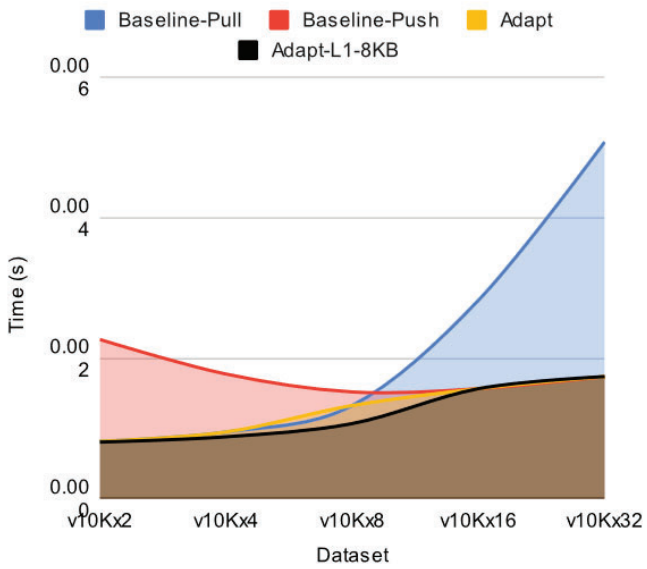


Figure 59: Execution Time Performance of Different Modes for SSSP

Next we tune the clock frequency to further improve the energy efficiency. The results for clock frequency of 0.1 GHz are shown in Figure 60. For graph size of 10K nodes with outdegree 32, the Adapt mode at 0.1 GHz has an energy efficiency of 7.27 GFLOPS/W, which is 1.7x higher than that of Adapt mode at 1.0 GHz (4.25 GFLOPS/W).

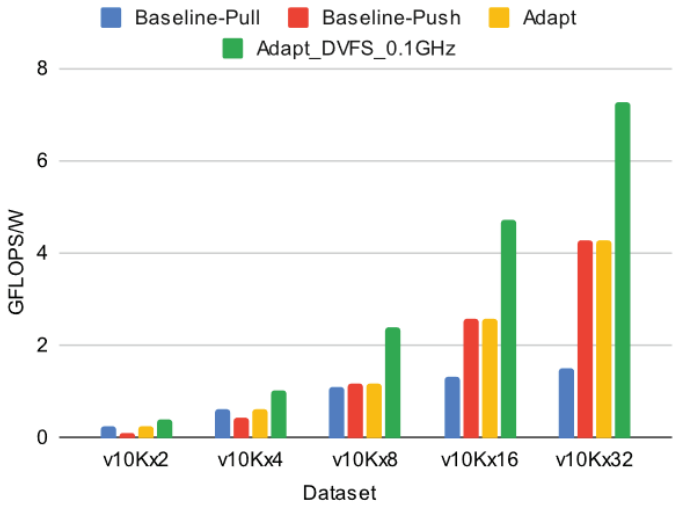


Figure 60: Energy Efficiency Performance of Different Modes for SSSP

Example Graph Convolution Network (GCN). It consists of aggregation phase and combination phase. We only consider aggregation phase since it is the only one that involves graph

traTransmuter v1.0l. The aggregation phase updates the information of the target node by aggregating feature vectors of all neighboring nodes. We implemented two modes of parallelisms as depicted in Figure 61. In node parallelism, each GPE works on separate nodes and handles the feature aggregation of all of its neighboring nodes. It enables feature data sharing if GPEs traverse to the same node. However, the node parallelism results in increase in cache misses if the feature vector is too large. The second mode uses hybrid parallelism which is a combination of tile-wise node parallelism, where LCPs traverse different nodes, and GPE-wise feature parallelism, where GPEs in the same tile work on aggregation of different columns of feature data.

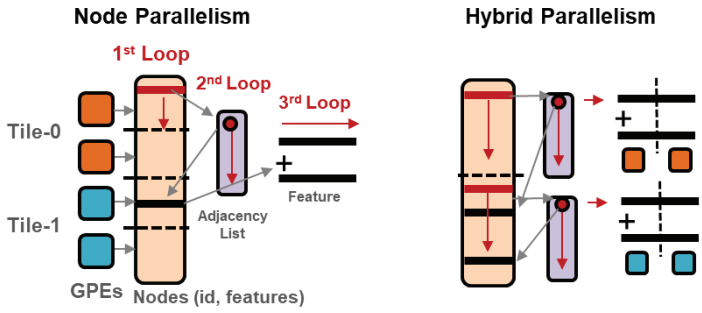


Figure 61: Two Parallelism of GCN-Aggregation

In GCN, feature size determines whether node parallelism or feature parallelism should be used. In our evaluation, the feature size threshold for switching |F| is around 100~140. However, we found the threshold |F| changes with graph sparsity. Sparser graphs have higher |F|, meaning for fixed feature size, optimal mode leans towards node parallelism.

Results: The static graph datasets to simulate a dynamic graph for GCN consists of 5 graphs with 10K nodes and varying out-degree of 2, 4, 6, 8 and 10. The feature size is fixed at 112. The Transmuter configuration is 4x16 (4 tiles with 16 GPEs per tile) with L1 banks of size 4 KB in shared cache mode and L2 banks of size 4KB L2 in shared cache mode.

We compare the performance of the following configurations: Node parallelism (Baseline-node), hybrid parallelism (Baseline-hybrid) and Adapt mode which chooses between node parallelism and hybrid parallelism. In addition, we increase the L1 cache to 8 KB to further improve the Adapt mode performance.

A comparison of the execution time of the different modes is shown in Figure 62. We see that use of the Adapt mode has a small advantage in reducing total execution time compared to

Baseline-node. With increase in L1 cache size to 8KB, the number of cache misses reduce, resulting in a decrease in the execution time, especially when the out-degree is larger.

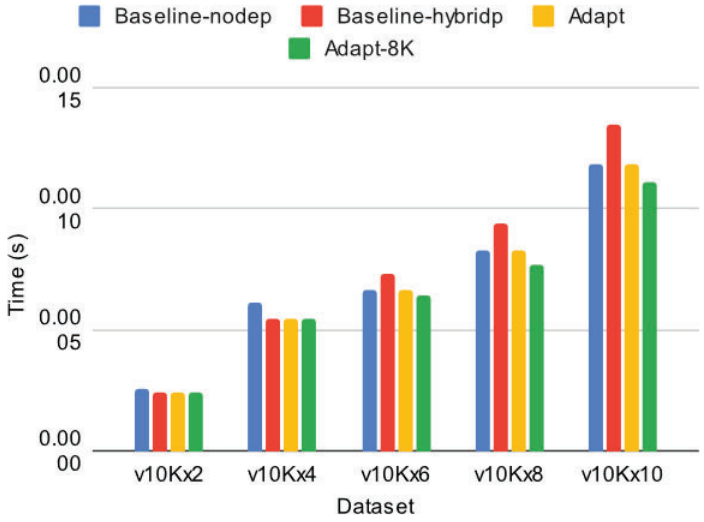


Figure 62: Execution Time Performance of Different Modes for GCN

We also tune the clock frequency to 0.1 GHz to further improve the energy efficiency of the Adapt mode. The results are shown in Figure 63. We see that for 10K nodes with out-degree 10, the energy efficiency increases to 3.9 GFLOPS/W when operated at 0.1 GHz from 1.39 GFLOPS/W when operated at 1 GHz – an increase of 2.8x.

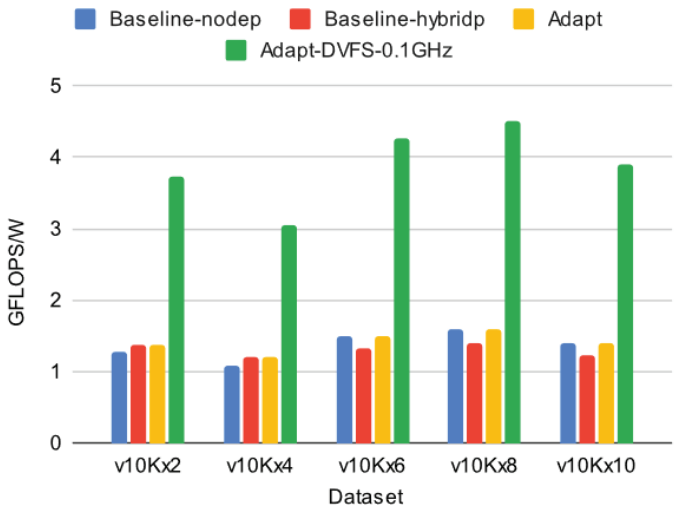


Figure 63: Energy Efficiency Performance of different modes for GCN

3.11 Radix-4 FAST Fourier Transform (FFT)

The radix-4 FFT algorithm uses $\log_4 N$ stages, where each stage consists of N/4 butterflies. Each butterfly computation takes four values as input and computes 4 output values. The computation can be expressed by a matrix-vector multiplication shown in Figure 64. Compared to Radix-2 FFT, Radix-4 FFT uses the same number of adds ($N \log_2 N$) but 75% of multiplications ($\frac{3}{8} N \log_2 N$).

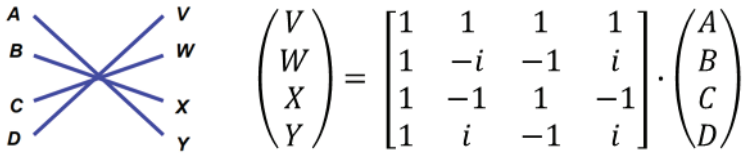


Figure 64: Butterfly in Radix-4 FFT Computation

Pipelined Radix-4 FFT Implementation

In the radix-4 FFT implementation on Transmuter, all GPEs in a tile process computations in a pipelined fashion. All butterfly computations in a stage are computed by the same GPE. Ping-pong buffers are used to store the input sequence and the intermediate values between stages. Figure 65 gives an example of a three-stage pipelined implementation with ping-pong buffers. Each GPE reads the input values from the buffer, computes the top part of the butterflies in the stage (as is indicated in the red circle), writes the output to the next buffer, and computes the next set (green circle). Each GPE communicates with its neighbor via the R2R queue to make sure the latest data is ready before the computation starts.

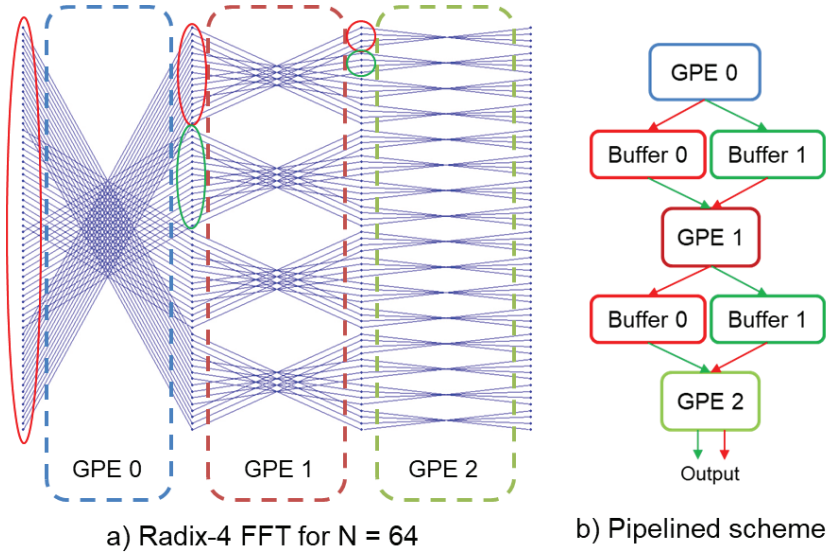


Figure 65: Pipelined Implementation of Radix-4 FFT for N=64 with Ping-Pong Buffer

The reconfiguration nodes to optimize implementation of Radix-4 FFT of different sizes are as follows.

- The number of active GPEs in a tile: FFTs of different sizes require a different number of stages and so different numbers of GPEs.
- L1/L2 configuration: Choose between cache and hybrid mode
- Storage mapping: Choose between cache and scratchpad for storing input sequence, twiddle factors, and intermediate values between stages
- Dynamic voltage and frequency scaling (DVFS): Choose optimal frequency/voltage setting

Results: We evaluated the performance of radix-4 FFT using 256-point and 1024-point FFT. The Transmuter architecture has 1 tile with 16 GPEs per tile. The L1 and L2 cache bank size is 4KB. For N = 256, 5 GPEs are activated -- 1 to load the input and the other 4 for computation. Similarly, for N = 1024, 6 (=5+1) GPEs are activated. For comparison, we use radix-2 pipelined implementation using R2R systolic design as the baseline. For the radix-2 implementation, 8 GPEs are activated for N=256 and 10 GPEs are activated for N=1024. In all cases, the cores run at 1.0 GHz.

We considered four mappings:

- Mapping 1: Shared L1 cache. GPE0 loads input using cache. Intermediate values and twiddle factors are in cache.
- Mapping 2: Hybrid L1 cache. GPE0 loads input using cache. Intermediate values are in SPM. Twiddle factors are in the cache.
- Mapping 3: Hybrid L1 cache. GPE0 loads input using cache. Intermediate values are in the cache. Twiddle factors are in SPM.
- Mapping 4: Hybrid L1 cache. GPE0 loads input using cache. Intermediate values and twiddle factors are in SPM.

The normalized execution time and energy efficiency performance are shown in Figure 66. We see that all mappings result in comparable performance with Mapping 4 (that uses SPM to store intermediate values and twiddle factors) having the best performance. Compared to baseline radix-2 implementation, Mapping 4 decreased the execution time by 19% for 256 FFT and 21% for 1024 FFT and increased the energy efficiency by 1.52x for 256 FFT and 1.54x for 1024 FFT.

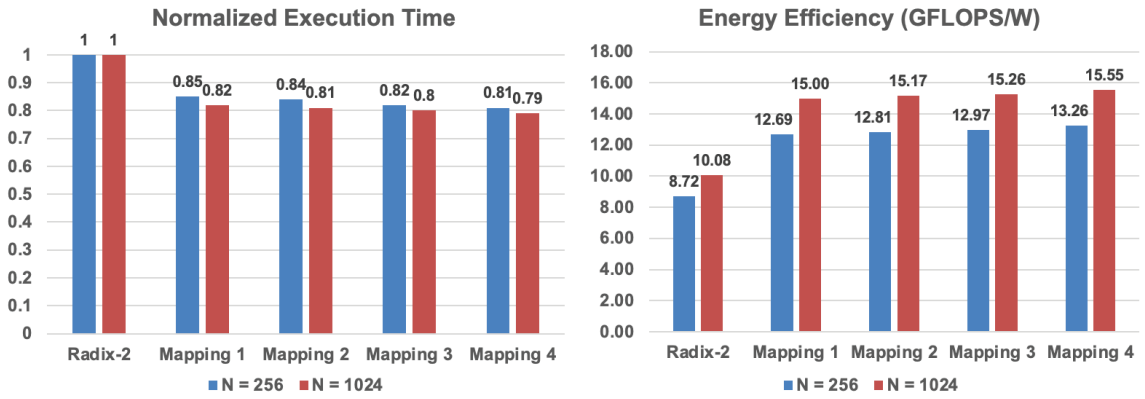


Figure 66: Performance Comparison: Execution Time and Energy Efficiency of different Radix-4 implementations

Next we applied DVFS to identify the setting with the highest energy efficiency, represented by GFLOPS/W, for radix-4 FFT. We used Mapping 4 and swept the core frequency from 0.1 GHz to 1.0 GHz. The results are shown in Figure 67. The results show that both 256 and 1024 FFT have the highest energy efficiency at 0.4 GHz core frequency. At 0.4 GHz, Radix-4 FFT achieves an optimal energy efficiency of 17.30 GFLOPS/W for N=256, and 20.67 GFLOPS/W for N=1024.

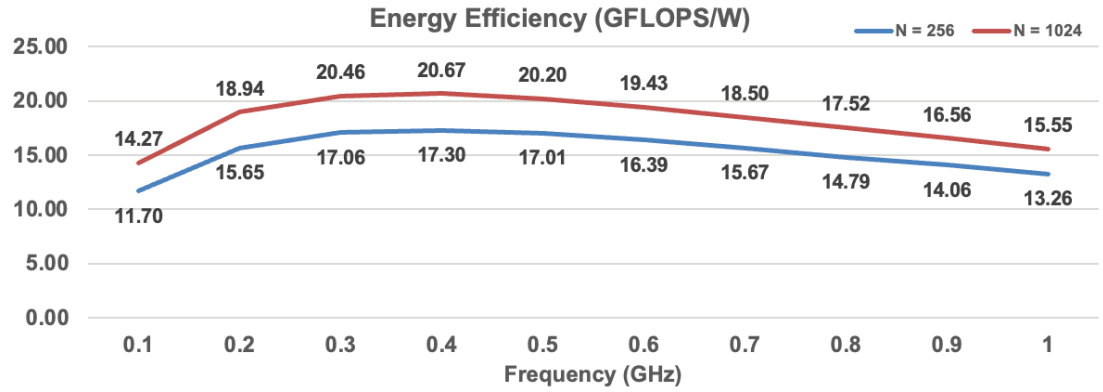


Figure 67: DVFS Evaluation of Radix-4 FFT

4 SECTION IV: TA-2 SOFTWARE DEVELOPMENT

4.1 Overview of Software Flow

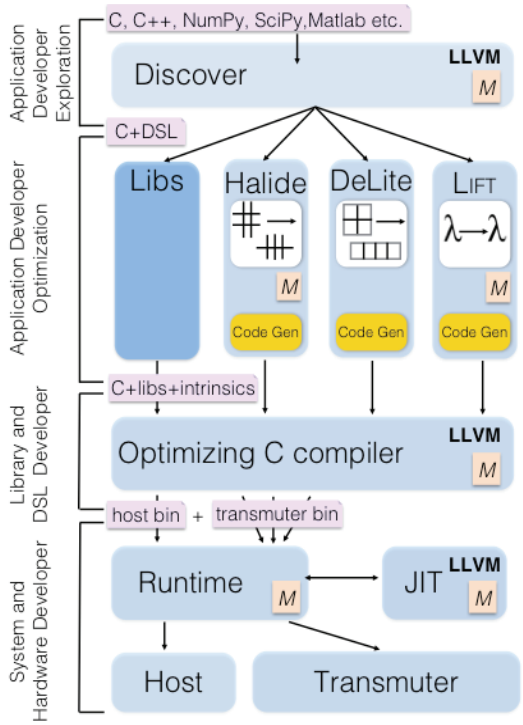


Figure 68: Programmer Tool Flow

Different DSLs/Libraries are exploited depending on fit to user code and performance. Multi-version code is passed to runtime, which selects the best during execution. Machine learning (peach box) is used at each stage to allow plug and play profitability models.

We will build a smart software stack based on our prior software and work [6,7,8] that automatically maps user code to Transmuter hardware. It will determine statically and dynamically the best hardware configuration for a particular program and runtime input based on our prior work. The hardware and software will be co-designed [9] to deliver efficient and flexible software defined hardware. There are two main themes in our technical approach (i) programmer productivity via abstraction and automatic optimization, and (ii) co-designed hardware and software based on machine learning.

4.1.1.1 C.2.1 Programmer Productivity

Programmer productivity is achieved by providing tools that minimize code modification while maximizing hardware performance. We achieve this by providing multiple entry points for code optimization and use machine learning as away of optimizing code and hardware. We will extend our work on idiom recognition and patterned parallel programming to achieve this.

Programmer tool chain. Rather than develop yet another programming language, even if it is an elegant DSL, we start off by developing a technique that targets existing DSLs, DISCOVER (see Figure 68). It reduces the programmer burden. This technique can automatically select parts of user legacy code and map it to existing DSLs, such as Halide[10], Milk[11], DeLite [12,13,14],

or Lift [15], or to existing specialized libraries, such as BLAS, cuBLAS, or cuSPARSE. DISCOVER achieves this by program synthesis, understanding the structure within the user program and then using a constraint system, maps it to an appropriate DSL e.g. stencils to Halide using LLVM. The initial version of the DISCOVER stage has been implemented in LLVM and is described in our paper [7]. It is a robust tool and has been tested on thousands of lines of existing C++ code. DISCOVER is aimed at early performance exploration by application developers. It takes their existing code written in C++, python, etc. and uses machine learning to determine the best DSL match. We also support users who wish to write directly in their favorite DSL or wish more control.

Once code is manually or automatically mapped to a DSL, we apply domain specific optimizations and hardware optimizations. DSL compilers (though not DeLite), have a transformation space that can be auto-tuned (Halide) or navigated (Lift) before code generation. DSL code generation normally produces low-level C. Here we will augment the backends to include low-level intrinsics that configure the hardware. This C+DSL stage is automatic or available to app developers interested in performance optimization.

The final layer contains an optimizing C compiler that is concerned with machine resources and has direct control over configuration. It, for instance, inserts code to direct the intelligent memory controller and crossbar. It produces a binary for the host and Transmuter hardware. Again, it uses machine learning in determining the best low-level optimizations. We have extensive work in this area [16,17,**Error! Bookmark not defined.**], implemented in LLVM which will act as a software starting point. In particular we have feature extraction tools and WEKA based machine learning models that can be used for rapid prototyping. This low-level compiler can be used by DSL and library developers to deliver specialized implementations that can be directly called from user code. ***Task 4*** can use this to hand implement specialized algorithms and then wrap them as libraries for later use by DISCOVER. The software stack will be supported by the Arm DS5 source debugger and profile tool.

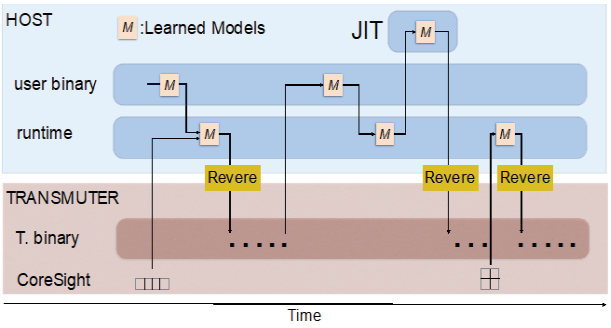


Figure 69: Runtime Timeline

User binary makes a call to the runtime to execute a kernel on Transmuter hardware. Depending on Coresight counters the runtime follows user request or chooses another code version or JIT recompiles. The runtime can reconfigure code and hardware during kernel execution if Coresight shows changed behavior.

Reconfigurable Runtime. Figure 69 demonstrates the control-feedback loop. The compiler generates several implementations, including settings for all configurable elements of the system, and execution begins. Based on our prior work [16], the actual implementation selected at runtime

depends on both static compiler knowledge and Coresight introspection, which is fed into a machine learning model (peach box) to determine what configuration to use.

The application object code running on the host makes calls to a host side runtime system expecting it to execute the selected code version on the Transmuter. Based on our prior work, the calls are intercepted by a runtime that can make decisions on whether to execute the code as is, or use a different implementation, based on Coresight information. If it believes all implementations are poor it can JIT recompile the code.

The Transmuter monitoring hardware periodically feeds back data to both the executing binary, and to the host runtime. When deemed appropriate by a machine-learned model, either the binary may make small reconfiguration changes on the fly, or, for more significant reconfigurations, the host side runtime will generate a new binary and/or configuration, or switch to an alternative precompiled library (versioning).

A key insight here is that by abstracting these mechanisms the system keeps control of reconfiguration, and can invoke it at well-defined "safe" points within the execution of Transmuter operations. This enables state-checkpointing to be enacted exactly when and where required. The actual policy employed by the runtime will be based on machine learning.

4.2 Programmability of Workflows

For all six workflows we were able to run the provided Python solutions out of the box with only trivial changes to import statements, as summarized in Figure 70, which records the number of lines of code changed in order to use our transpy library and stack. In addition, for LGC, we produced a second version, discussed in detail in the following section, in which we made further changes to extend "coverage", ie the proportion of overall execution delegated from the host to the transmuter.

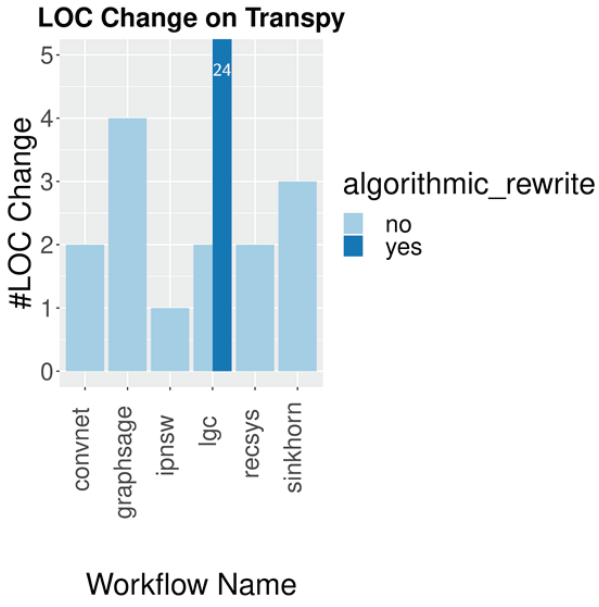


Figure 70: Analysis of the Software Framework in Terms of Number of lines of Code Changed

These workflow versions generated the coverage and GOPS/W presented in Figure 71. Coverage indicates the fraction of work in the overall end-to-end workflow which was delegated. For example, a coverage figure of 70% indicates that 70% of the work done (or in terms of execution time, 70% of the execution time on a host-computes-everything sequential run) would be delegated. This creates an Amdahl's Law situation, in that the overall performance achievable will be constrained by the coverage, both in terms of execution time (as in a normal Amdahl's Law) and energy efficiency (where overall GOPS/W are similarly constrained by an Amdahl-for-Energy Law, with two GOPS/W rates) . GOPS/W in Figure 71 are only for the code which would be executed on the transmuter. The full performance tables include both the transmuter and host system. Notice that by rewriting 24 lines of the original LGC were able to roughly double its coverage. Interestingly, its GOPS/W figure for the TM-only portion is unchanged: in other words, the new version is maintaining the same rate for a larger proportion of the underlying work, which will be reflected in the improved overall GOPS/W for the whole execution.

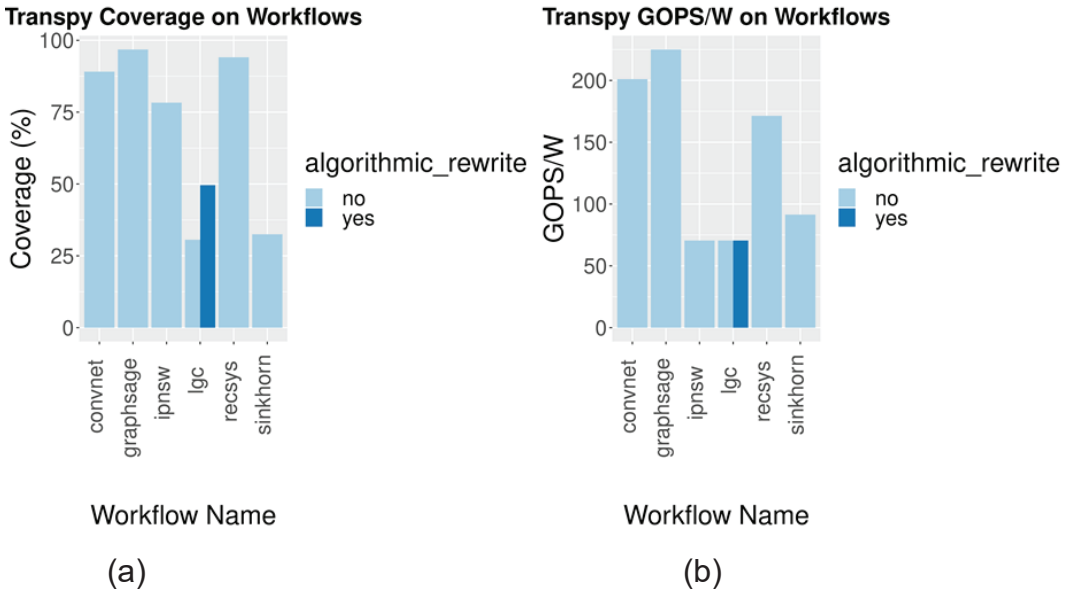


Figure 71: (a) Coverage and (b) GOPS/W Predicted for Each Workflow

We observe that the host execution time figures reported in the summary tables and spreadsheet are sometimes considerably larger than for the corresponding executions reported in the CMU-SEI reference data. Apart from minor differences in host platform, this is explained by inefficiencies in our current python stack, where there is considerable internal copying of data, during data-marshalling. This has not been our primary focus during phase 1, in which we have focused on transmuter code coverage, and will be optimized away in subsequent releases. It's important to note that this is all happening in the fraction of the workflow which will not be delegated to the accelerator (i.e the inherently sequential residual code). We presume that the hand-coded CMU-SEI implementations are already avoiding this overhead in their own sequential portion. As expected, the effect is particularly strong in the workflows with relatively large data usage (hence our copying is expensive), and relatively little computation per unit data (hence the impact of unnecessary copying is not masked). For example graphsage, IPNSW and convnet are vulnerable to this effect.

4.3 TA2 Notes on Programmability - the LGC Case Study

A simple inspection of the original LGC code (version with two lines changed, to import our transpy), revealed two instances of “inappropriate” coding style, in which the programmer has used a loop to traverse a data structure. In each case, replacing the loops with equivalent bulk-data transpy operation captured the same computation, but with simper code and improved coverage. The necessary code changes are listed below, with a third simple knock-on change required to complete the adaptation.

The first instance transformed the vertex map calculation

```
        for node_idx in frontier:
            p[node_idx] += (2 * alpha) / (1 + alpha) * r[node_idx]
            r_prime[node_idx] = 0
```

to the following

```
vertexmap = ((2 * alpha) / (1 + alpha)) * r * frontier
p += vertexmap
r_prime = r * (np.where(Frontier==1,0,1))
```

The second instance transformed the edge map calculation from

```
for src_idx in frontier:
    neighbors = adj.indices[adj.indptr[src_idx]:adj.indptr[src_idx + 1]]
    for dst_idx in neighbors:
        update = ((1 - alpha) / (1 + alpha)) * r[src_idx] / degrees[src_idx]
        r_prime[dst_idx] += update
```

to the following code

```
#Rewrite edge-map loop using transpy
f = ((1 - alpha) / (1 + alpha)) * r / degrees * frontier
edgemap = csg.T.dot(f)
r_prime += edgemap
```

Finally, the frontier update phase was adapted from

```
frontier = np.where((r >= degrees * epsilon) & (degrees > 0))[0]
```

to

```
#New frontier for next iteration
frontier = np.where((r >= degrees * epsilon) & (degrees > 0),1,0)
```

These are good examples of the coding style promoted by our programmer guidance and training documentation: try to use bulk operations rather than explicitly coded loops. We note that in phase 3 we plan to use our programming idiom recognition technology to discover and transform such instances in legacy code.

4.4 SparseAdapt: Runtime Control for Sparse Linear Algebra on a Reconfigurable Accelerator

We developed a post-silicon adaptation framework as part of the software runtime of the proposed software-defined hardware (SDH) system. This framework is called SparseAdapt and it adapts to the evolving nature of data and code that naturally occurs as part of many algorithms involving sparse data structures, such as graph traTransmuter v1.0l and algebraic solvers. SparseAdapt uses a set of offline-profiled training examples to learn the mapping of hardware performance counter data to the best set of configuration parameters given the counters.

SparseAdapt then uses periodic hardware telemetry during the execution of any given application to tune the hardware, based on the observed program behavior (illustrated in Figure 72). It can be programmed to optimize for either the energy efficiency or power-performance efficiency for the program run. Under the hood is a predictive model composed of an ensemble of decisions trees, one per configuration parameter that can be tuned; for this effort we considered tuning the clock/voltage (dynamic voltage frequency scaling), dynamic cache capacities, L1 resource sharing mode, L1 on-chip memory type, and prefetcher aggressiveness.

Compared to the static (non-reconfiguring) Transmuter configuration that is best (on-average) for the sparse computation domain, a system equipped with SparseAdapt achieves a 1.6× improvement in performance-per-Watt in the Energy-Efficient mode of operation, or the same performance with 23% lower energy in the Power-Performance mode, for SpMM across a suite of real-world inputs. For SpMV with scratchpad configuration, our framework demonstrates 1.8× better performance for ~10% less efficiency compared to this static configuration. Finally, compared to the state-of-the-art approach for dynamic runtime control, our proposed mechanism outperforms by 1.7-2.8× in performance and 1.1-2.0× in energy-efficiency. This work was published at the IEEE/ACM International Symposium on Microarchitecture (MICRO).

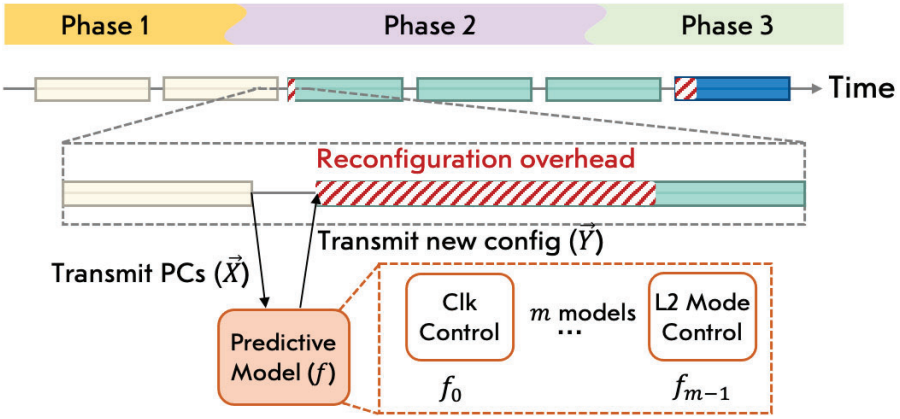


Figure 72. Illustration of the Proposed SparseAdapt Scheme

4.5 Programable Prefetcher

In this section, we describe the design of a programmable prefetcher designed for graph workloads using hardware/software co-design. The software communicates program semantics information, i.e., data structure layout in memory and algorithmic traversal pattern, to the hardware. We present a compact representation called Data Indirection Graph (DIG) to communicate this information. Two techniques can be used to achieve this – manual programmer-inserted API calls and automatic code generation using compiler analysis. This knowledge is used by a hardware prefetcher to prefetch according to an irregular algorithm's traversal pattern. We show an average performance improvement of 1.5x compared to the state-of-the-art hardware prefetchers.

IV.I. Programming Model

In this section, we first present DIG representation and how to construct it.

IV.I.I. Data Indirection Graph (DIG) Representation

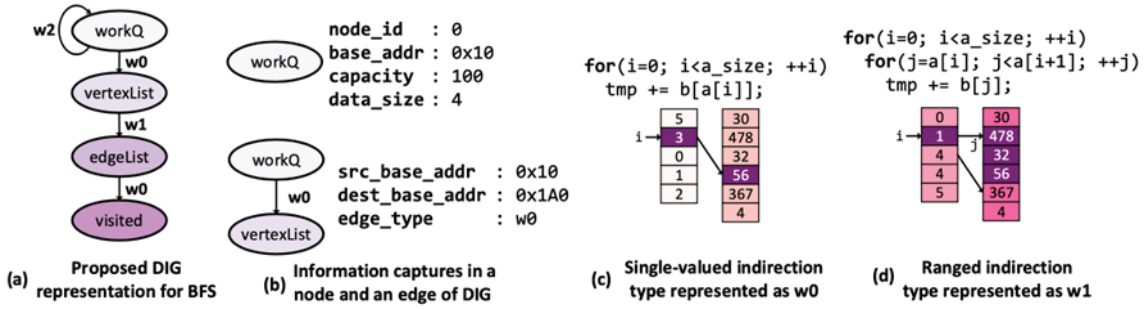


Figure 73: Proposed Data Indirection Graph (DIG) Representation

(a) example representation for BFS, (b) data structure memory layout and algorithmic traversal information captured by a DIG node and a weighted DIG edge respectively; two unique data-dependent indirection patterns supported by our system - (c) single valued indirection, and (d) ranged indirection.

We make the key observation that the graph processing algorithms use two specific types of data-dependent indirection patterns to index from one data structure to another. We can form a basis set using these two patterns and create a combination set of these patterns that spans the entire indirection pattern of a graph traversal algorithm. With this insight, we propose a graph representation, which we call a data indirection graph (DIG), to capture the relationship between data structures for graph algorithms.

In a DIG, each node represents a data structure (e.g., the visited list in BFS), and each directed weighted edge represents a data-dependent access. Figure 73 shows an example DIG representation for the BFS algorithm with workQueue, vertexList, edgeList, and visited data structures. Nodes store meta data information about the data structured that include an abstract node ID, base address, capacity, and data size. Two type of data-dependent indirection patterns include single valued indirection (see Figure 73c) and ranged indirection (see Figure 73d) that we annotate in the DIG using an edge weight (w0 for single-value and w1 for ranged indirection). A special type of edge called trigger edge is added to the data structure from where the prefetch sequence is triggered. The figure shows the workQueue with trigger edge with a weight of w2 that dictates actions to perform in order to initiate a prefetch sequence.

IV.I.II. Manual DIG Construction

```
1: int BFS(FILE* inputGraph, vtxID source)
2: {
3:   Graph g = readGraph(inputGraph);
4:   queue<vtxID> workQueue(g.numNodes());
5:   vtxID** vertexList = (vtxID**) malloc(g.numNodes()+1);
6:   vtxID*  edgeList   = (vtxID*)  malloc(g.numEdges());
7:   vtxID*  visited    = (vtxID*)  malloc(g.numNodes());
8:   populateDataStructures(g, vertexList, edgeList, visited);
9:   registerNode(&workQueue,  g.numNodes(),   4,  0);
10:  registerNode(vertexList,  g.numNodes()+1, 4,  1);
11:  registerNode(edgeList,    g.numEdges(),   4,  2);
12:  registerNode(visited,     g.numNodes(),   4,  3);
13:  registerTravEdge(&workQueue, vertexList,  w0);
14:  registerTravEdge(vertexList, edgeList,    w1);
15:  registerTravEdge(edgeList,   visited,     w0);
16:  registerTrigEdge(&workQueue, w2);
17:  workQueue.enqueue(source);
18:  […]
```

Figure 74: Manual Code Insertion by the Programmer

Assuming that the programmer is cognizant of the key data structures and traversal algorithms used in the application, they can add simple API calls in the application source code to construct the DIG representation. Figure 74 presents these modifications, where three unique API calls are used to annotate the DIG. The calls to register nodes and edges are registerNode and registerTravEdge and registerTrigEdge, respectively.

IV.I.III. Automatic DIG Construction

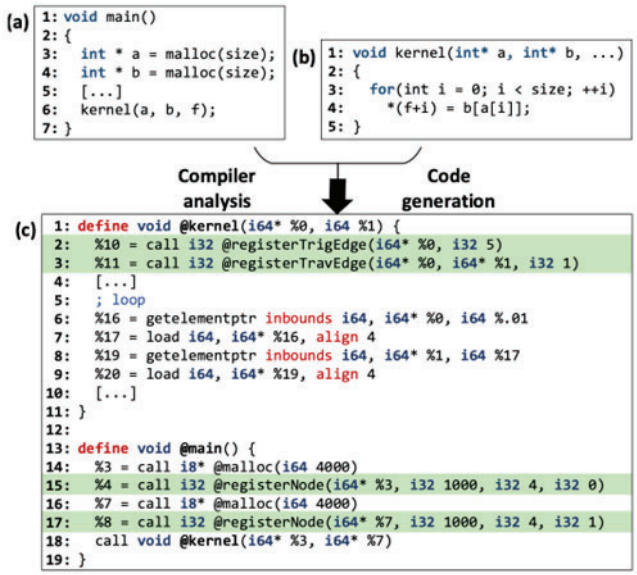


Figure 75: Automatic Code Insertion by the Compiler

Identifying indirections in non-trivial programs can be a complicated task for the programmer, often requiring in-depth application knowledge usually held by experts. Our compiler alleviates this manual work by automatically identifying these indirections and transforms the program by annotating it with calls to the prefetcher API.

First, our compiler analysis extracts information required for node registration from allocations. Figure 75c shows two node registrations, each using information from the immediately preceding malloc calls. Next, by tracking the use of these nodes, it extracts edge information and detects their associated indirection patterns. Figure 75b contains a single-valued indirection in the form of a load from b[a[i]] (line 4), which corresponds to the LLVM IR in lines 6-9 of Figure 75c. As the base addresses of these two arrays form the edge between the nodes, our pass extracts them and uses them in the registerEdge function along with the final argument that specifies the type of edge being registered - in this case, a single-valued indirection. Our code generation pass places the edge registration calls as soon as all of the required arguments have been defined. In Figure 75, the pointers to the arrays are passed into the kernel as arguments, allowing the edges to be registered at the start of the function (lines 2-3). Ranged indirection can be identified in a similar manner.

IV.II. Hardware Design

This section briefly talks about the hardware design of our prefetcher.

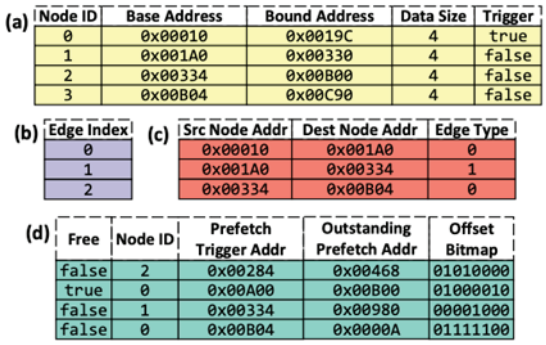


Figure 76: Proposed Data Indirection Graph (DIG) Representation
(a) example representation for BFS, (b) data structure memory layout and algorithmic traversal information captured by a DIG node and a weighted DIG edge respectively; two unique data-dependent indirection patterns supported by our system - (c) single valued indirection, and (d) ranged indirection.

IV.II.I. Prefetcher Memory Requirements

Figure 76 (a-c) show three prefetcher-local memory structures to store a DIG representation. As described in Section IV.I.I, the node table and the edge table store properties of nodes and edges respectively. The base address, number of elements, and data size of each node specified by software are converted into base and bound addresses by our run-time library and then stored in the node table. Additionally, we use an edge index table to find outgoing edges from a DIG node, which mimics the software vertex list in hardware.

Additionally, the prefetcher also uses a prefetch-status handling register (PFHR) file to keep progress of issued prefetch requests as well as to make the prefetcher non-blocking. As show in Figure 76d, the PFHR file stores the abstract node ID from where the prefetch is issued, virtual

address from where the prefetch sequence starts, the outstanding prefetch request, and a bitmap of byte addresses to keep track of which addresses are sent for prefetches from a cacheline.

IV.II.II Hardware Control Flow

Figure 77(a) shows the design and interface of the hardware prefetcher with the rest of the system. Once the prefetcher is programmed with the DIG representation, it snoops on load requests from the core pipeline to the L1D cache and waits for a demand request that falls into address ranges of the data structure with the trigger edge.

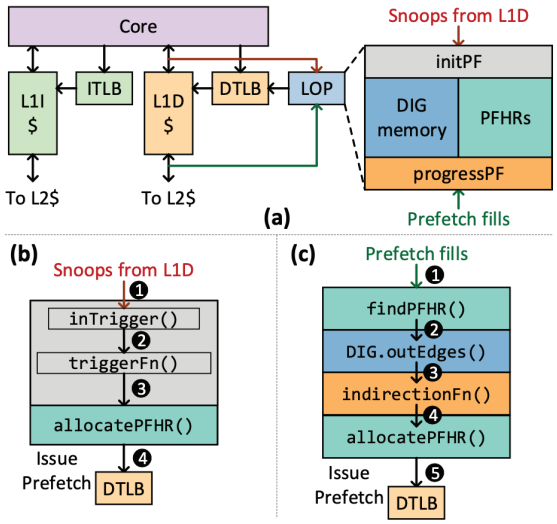


Figure 77: Prefetcher Design and Control Flow

(a) Prefetcher hardware design and its interface with the rest of the system, (b) steps to initialize a prefetch sequence on a demand load to the trigger array, and (c) steps to advance a prefetch sequence on a prefetch fill.

Prefetcher state machine goes through two phases for issuing prefetch requests. The first prefetching phase is called the initiate prefetch sequence. Figure 77(b) shows the series of steps involved in this phase. Once a load address is observed within the address bounds of the node with the trigger edge, the trigger function is called, which returns a set of addresses to prefetch. For each address in this set, a new PFHR is allocated. A prefetch request is then issued via the DTLB, which translates the address. Once the request is issued, prefetcher state machine waits for outstanding requests to get serviced and responses are handled in the second phase. Figure 77(c) shows the second prefetching phase, called the advance prefetch sequence, which receives data from prefetch fills and continues the prefetch sequence. Upon prefetch fills, we first find the PFHR entry that is keeping track of this request by performing a CAM look-up in the PFHR file. After identifying the correct PFHR, we get its source node ID and find its outgoing edges and destination node IDs by indexing into the edge and edge index tables. Further prefetch requests are generated using the indirection function stored in the edge table. This process continues until we find the leaf node in the DIG after which, we do not track the prefetch sequence anymore using the PFHR file.

IV.III. Evaluation

To evaluate the potential of our design, we first implement our prefetcher design using a x86-based CPU simulator called Sniper. We use 4-core system for evaluation having a three-level cache hierarchy similar to commercial processors. We use GAPBS – hand-optimized version of graph algorithms and real-world large-scale power-law graph data sets for evaluation.

Figure 78 shows the performance comparison of various prefetchers with our work. We see that pir prefetcher outperforms a non-prefetching baseline and a GHB-based G/DC [1] data prefetcher by 2x on average. GHB-based G/DC is known to predict inaccurate prefetch addresses for irregular memory accesses due to the lack of spatial locality. Its performance is similar, and in some cases worse than a non-prefetching baseline as it can pollute the cache hierarchy with useless data.

Our prefetcher also outperforms the Ainsworth and Jones' prefetcher [2] by 1.8x mainly because their look-ahead distance calculation does not work well for the benchmark implementations we use. We use a more robust prefetching algorithm that can dynamically adapt to changing machine states, i.e., cache contents and variable load latency. Compared to DROPLET [3], our work achieves a 1.7x speedup on average for two reasons. First, DROPLET only prefetches a subset of data structures, i.e., edge list and visited list-like arrays, compared to ours, which

prefetches other graph data structures as well. Second, we notice that DROPLET MPP misses several prefetching opportunities, since it can only trigger prefetches from requests serviced from DRAM, while much of the prefetched data is already present in the cache hierarchy. Our proposal also achieves an average speedup of 1.6x compared to IMP [4], since IMP can only detect streaming accesses to data structures to perform A[B[i]] type prefetching and it only supports up to two levels of indirection.

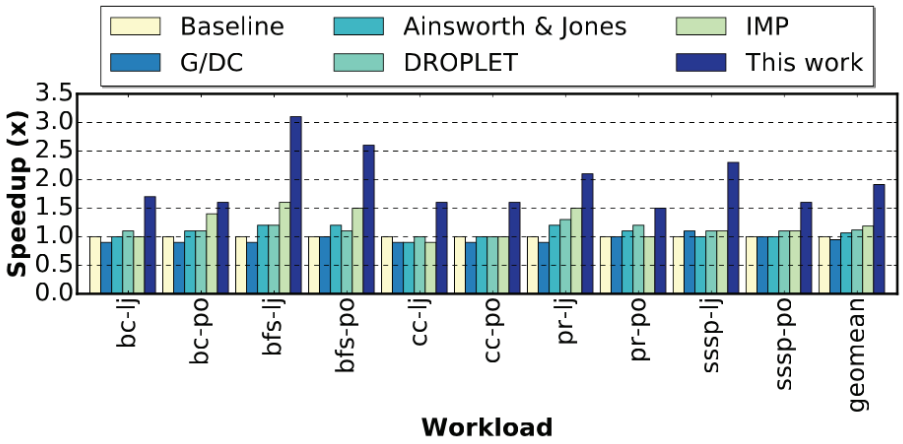


Figure 78: Performance Comparison of a Non-prefetching Baseline, GHB-based G/DC Prefetcher [1], Ainsworth and Jones' prefetcher [2], DROPLET [3], IMP [4], and this Work

Higher is better.

IV.III. Integration with Transmuter

While we have only demonstrated the effectiveness of our prefetching mechanism with CPU-based systems, we will also integrate our ideas into Transmuter in the near future. Specifically, the software-defined aspect of Transmuter makes our prefetching especially application because of native support to communicate information between software and hardware. The DIG construction and prefetcher programming can exploit this hardware/software contract. We envision a private instance of our prefetcher attached to each of the GPEs of Transmuter that can snoop load requests going to L1D caches via crossbar. To have a global view of load requests irrespective of Transmuter configuration, we can hook up the prefetcher along the address request lines out of GPEs before the request is routed via the crossbars. Since graph processing workloads suffer from memory stalls, we believe that our prefetcher can successfully alleviate the memory stalls to improve performance.

References:

[1] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in10th International Symposium on High Performance Computer Architecture (HPCA'04), Feb 2004, pp. 96–96.
[2] Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in Proceedings of the 2016 International Conference on Supercomputing, ser. ICS '16. New York, NY,USA: ACM, 2016, pp. 39:1–39:11.
[3] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Feb 2019, pp. 373–386.
[4] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Dec 2015, pp. 178–190.

4.5.1 FFT Compiler for Legacy Code Mapped to Accelerators

The University of Michigan, Arizona State University, and the University of Edinburgh, under the DARPA MTO Software-Defined Hardware program, has developed a compiler infrastructure, called FACC – a compiler that matches legacy code to FFT accelerators using IO behavioural equivalence and program synthesis. We evaluate it using 20 Github projects from others and show in Figure 79 and Figure 80 the performance improvement of applying FACC to the Transmuter relative to the performance achieved on 2 existing accelerators: the NXP PowerQuad and the Analog Devices FFTA. In both cases an appropriate host CU is used as a comparison baseline. For the low-power cases we use an ARM M33 while in the high-performance case we use an ARM Cortex A5. In both cases Transmuter outperforms its accelerator counterpart. This work was published at PLDI [1].

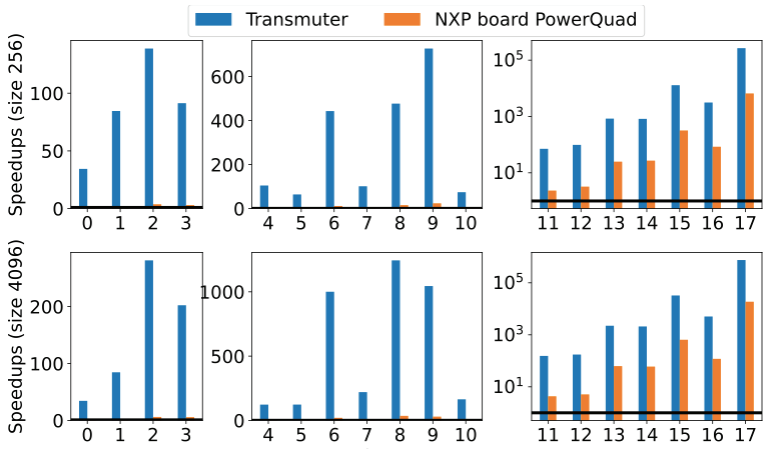


Figure 79: Speedup of using FACC on Accelerators vs Running on a ARM Cortex M33 CPU Baseline

The 2 accelerators are Transmuter and an NXP PowerQuad. The NXP accelerator improves performance for naïve implementations, but Transmuter shows improvement across all Github projects.

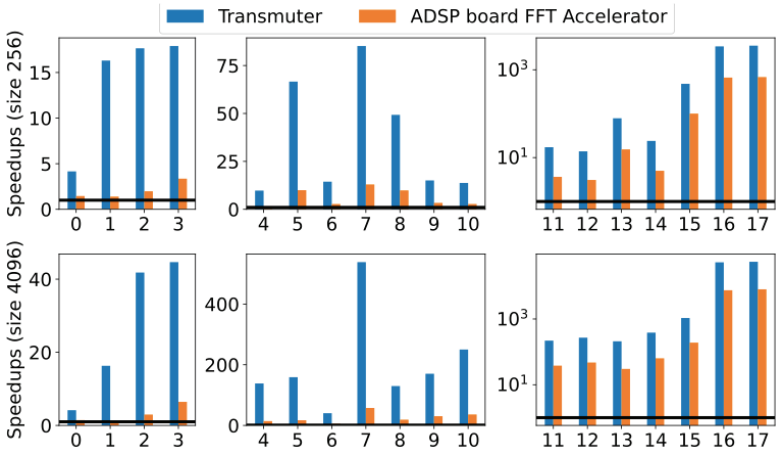


Figure 80: Speedup of using FACC on Accelerators vs Running on an ARM Cortex A5 CPU Baseline

The 2 accelerators are Transmuter and an Analog Devices FFT accelerator. The Analog Devices accelerator improves performance in the majority of cases, but Transmuter shows increased improvement across all.

[1] *Woodruff, Armengol-Estapé, Ainsworth, and O'Boyle. "Bind the gap: compiling real software to hardware FFT accelerators." In Proc. of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022). Pp. 687–702 .*

5 SECTION VI: APPENDIX OF PHASE 1 ALGORITHIM STUDY RESULTS

5.1 GraphSAGE Analysis – Phase 1

Overview: GraphSAGE is a neural network model used on graph-structured data. The network is used to predict the properties of a given node based on the known properties of its one-hop neighbors as well as the two-hop neighbors derived from the one-hop neighbors.

5.1.1 Architecture Overview

In general, the GraphSAGE algorithm is broken down into four parts:

- Embedding: The input node index is used to generate random sample of one-hop and two-hop neighbors. These neighbors are then passed through an embedding layer to extract the features associated with each sampled node.
- Forward Pass (inference): The neural network is organized as shown in Figure 81. There are five linear layers (linear1, linear2, hid1, hid2, out) as well as two pooling layers (mean2, meanHid1) and a concatenation step.
- Backpropagation: The prediction error is propagated backwards through the network to calculate the weight gradients of each hidden layer.
- Weight Update: A simple SGD optimizer is used to update the weights based on the weight gradients generated in the backpropagation step.

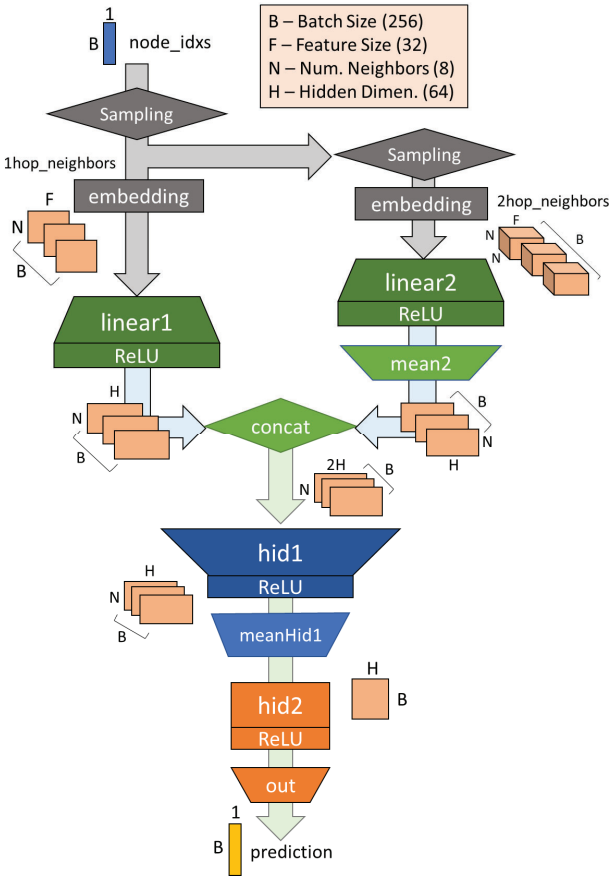


Figure 81: GraphSAGE Network Overview

The baseline configuration of Transmuter is 4 tiles, 16 GPEs/tile, 4 KB L1 cache and 64 KB L2 cache. Both L1 and L2 are configured in **private cache mode**. The GraphSAGE network is trained with a batch size of **256**, which was partitioned evenly across the GPEs. While the number of neighbors for the provided DARPA workload is 12, majority of our analysis was conducted with the number of neighbors at 8 for faster simulation.

To ensure there is no data collision between the tiles, the batch size must be a factor of NUM_TILES*16 for tensors of int and float values (and larger datatypes). This is an alternative to the use of synchronization primitives (which slows us down) or padding the data (which is memory intensive) to ensure data coherence across the different tiles.

Due to the constraints of the simulation infrastructure, we are running only one epoch for DARPA's default input arguments for one iteration (one batch). Since the next iterations and epochs do the exact same calculations on different data sets with the exact same size, the execution time of full workload can be estimated by scaling our execution time by epochs*graphsize/batchsize. The graphsize of the training workload is 2143, so for the batchsize of 256, we are simulating 4% of the total DARPA workload.

5.1.2 Performance Results

Kernel Breakdown

To better understand the GraphSAGE workload, we first look at the runtime breakdown of each kernel as shown in Figure 82. The backpropagation portion, which comprised 77% of the total workload, was further broken down into the individual layers of the neural network. We see that backpropagation dominates the runtime, due to the gradient computation of its hidden linear layers hid1, linear1, and linear2 that operate on 3D and 4D tensors.

The embedding kernel is part of the initialization phase which executes solely on the LCP0 of the transmuter. Once the data initialization and embedding is complete, the rest of the workload is distributed across the entire transmuter.

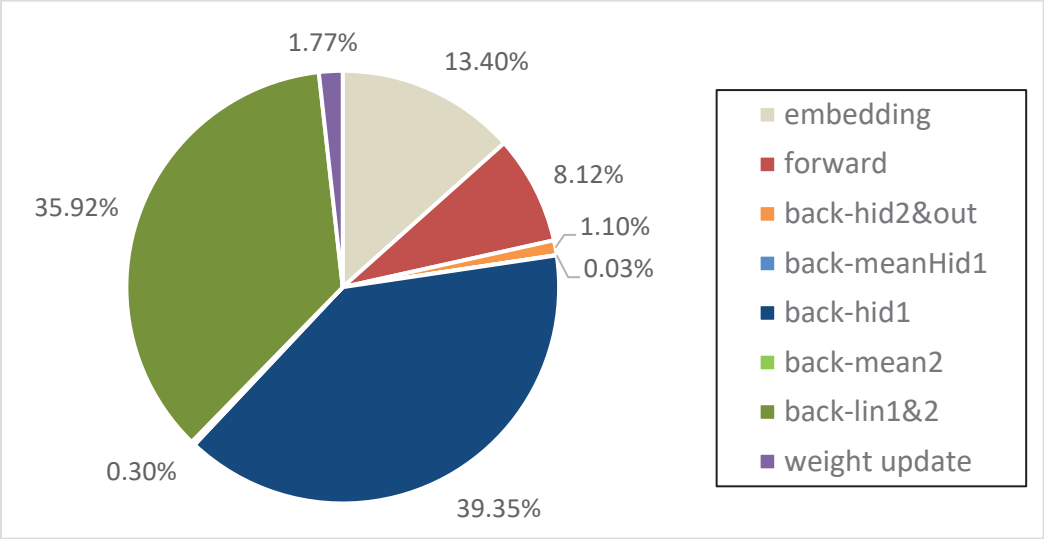


Figure 82: GraphSAGE Kernel Breakdown for Baseline Transmuter

Cache Size Variation

We look at how variation in L2 cache size (4, 16, and 64KB) impact the performance of the Transmuter in terms of both performance and power, as shown in Figure 83. The power is estimated with 14 nm technology and the clock frequency of the whole system is 1 Ghz. For these data, the "Backpropagation" includes both the backpropagation phase and the weight updating phase.

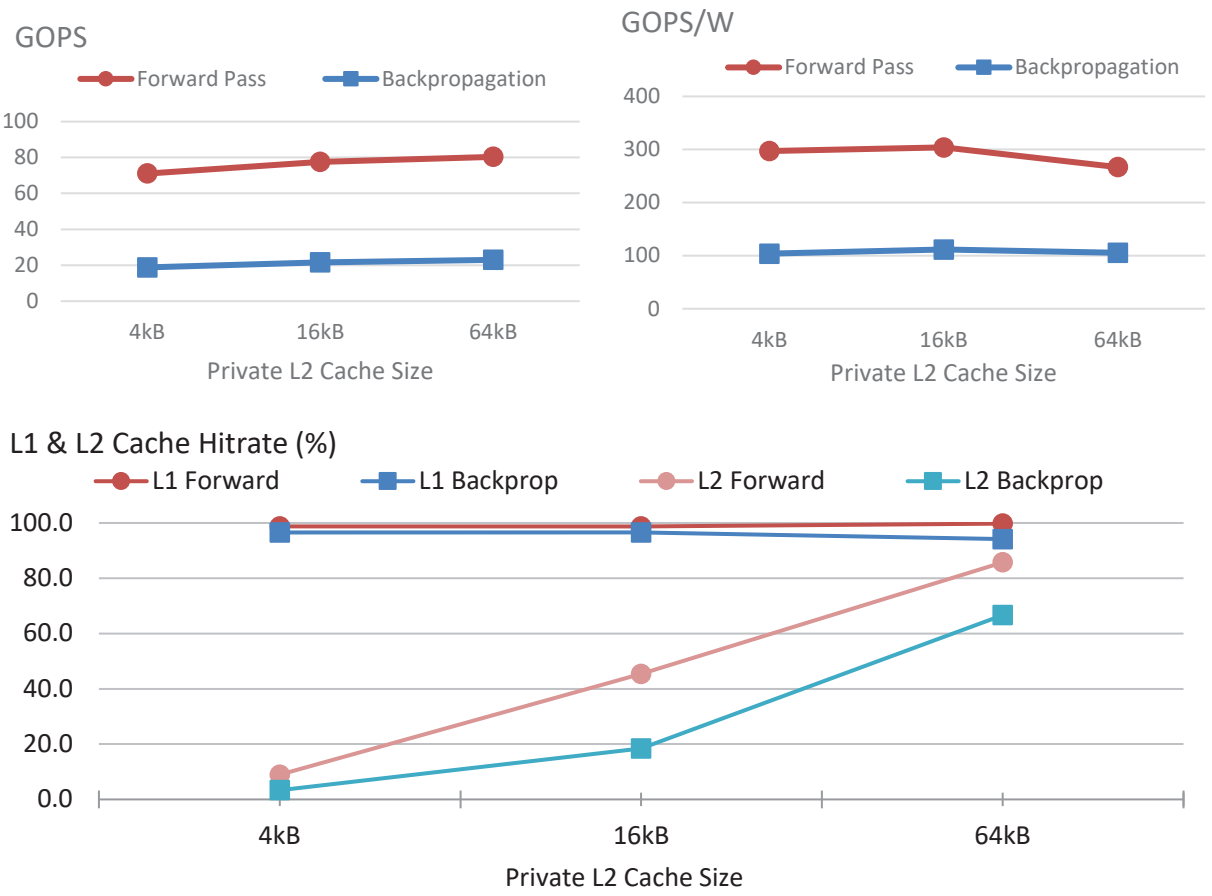


Figure 83: Performance Scaling at Different L2 Cache Sizes

The Transmuter performance increases with the L2 cache size, as shown by the higher GOPS and L2 cache hit rate at 64KB. However, the 16KB configuration is more optimal when power is taken into account, as shown by the decrease in GOPS/W at 64KB compared to 16KB.

5.1.3 Graphsage SDH Performer Performance Tables

As shown in Table 13, the best performance of Transmuter setup for GraphSAGE is private cache mode with 64KB L2 cache bank size if optimized for performance only, and 16KB for overall efficiency.

Table 13: Performance Table of Different Cache Configurations. BatchSize = 256, Num. Neighbors = 8

Configura-tion	GOPS/W	GOPS	GFLOPS/W	GFLOPS	Total Power (W)	Total Time (s)	L1 hit rate (%)	L2 hit rate (%)
4x16 L2: 4kB Private	122.17	20.04	10.74	1.76	0.16	0.30	96.70	3.70
4x16 L2: 16kB Private	130.62	22.53	11.46	1.98	0.17	0.27	96.70	20.40
4x16 L2: 64kB Private	117.45	22.74	12.81	2.48	0.19	0.26	94.50	67.60
4x16 L2: 4kB Shared	77.30	12.28	7.89	1.25	0.16	0.48	94.20	0.80
4x16 L2: 16kB Shared	81.20	8.07	7.98	0.79	0.19	0.79	94.50	2.10

Running this optimal Transmuter configuration on the DARPA dataset with number of neighbors at 12 results in Table 14. The initialization phase consists of loading the DARPA dataset as well as the embedding phase. This phase is performed only on a single LCP while the remaining system remains idle to conserve energy. It is computed for every batch and costs 0.067 seconds.

Table 14: End-to-End Performance Table for DARPA dataset (N=12) with Optimal Transmuter Configuration

Configuration (N=12, B=256)	GOPs/Watt	Execu-tion time (s)	Total energy (J)	Simu-lated/ es-timated cycles	L1/L2 Cache hit rates (%)	Percentage of system simulated
4x16 L2: 16kB Private	72.69	0.98	0.16	978325169	93.2% / 1.56%	3,98%

Initialization			Main Kernel			End-to-End		
GOPs/Watt	Execu-tion time (s)	Total en-ergy (J)	GOPs/Watt	Execu-tion time (s)	Total en-ergy (J)	GOPs/Watt	Execution time (s)	Total en-ergy (J)
174.752	0.067	0.001	72.017	0.911	0.157	72.69	0.98	0.16

5.1.4 Scalability

Software

To determine the scalability of our GraphSAGE implementation, we compared how the execution time scaled with increased the batch size as shown in Figure 84. Since there is no communication between the GPEs once the workload is distributed, we see the execution time of the workload almost doubling each time the batchsize is doubled.

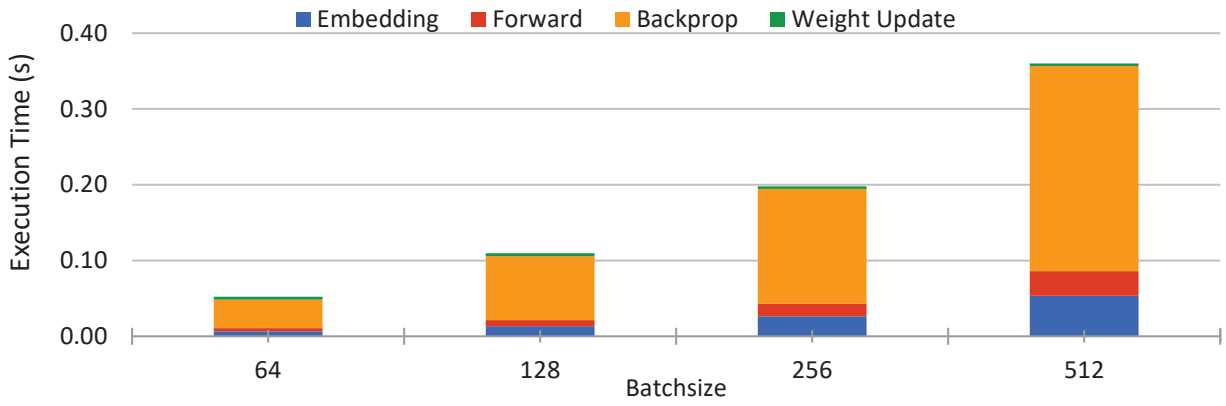


Figure 84: GraphSAGE Runtime at Different Batch Size

The number of neighbors (N) is another key variable for the GraphSAGE workload, as it determines how many neighbors get sampled, which results in quadratic increase in the number of nodes chosen for the two-hop neighbors. **Figure 85** shows increase in number of neighbors resulting in exponential increase in execution time, as expected.

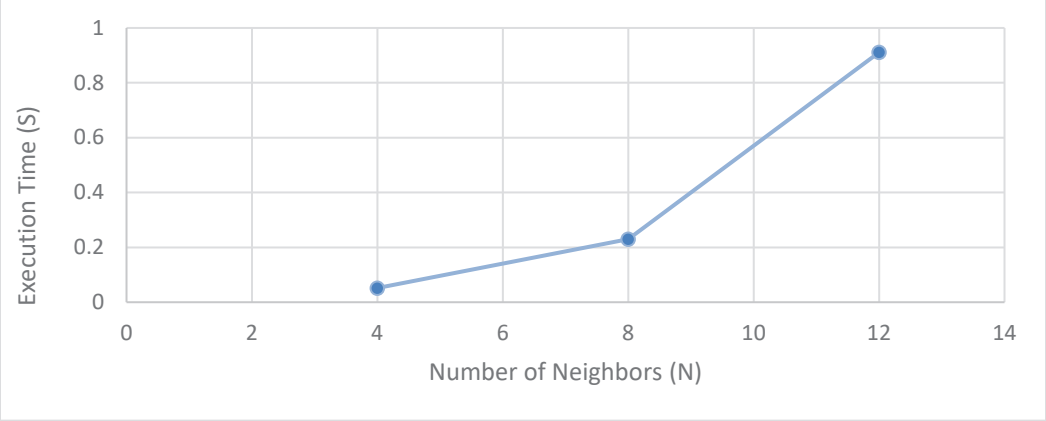


Figure 85: GraphSAGE Runtime at Different Number of Neighbors

Hardware

The graph in Figure 86 shows how Transmuter performance scales as the number of tiles and GPEs are increased. Due to the highly parallel nature of the algorithm, GraphSAGE performance scales linearly with the increase in tiles count. Despite the increase in processing elements, the synchronization overhead remains low thanks to the efficient design of the crossbar.

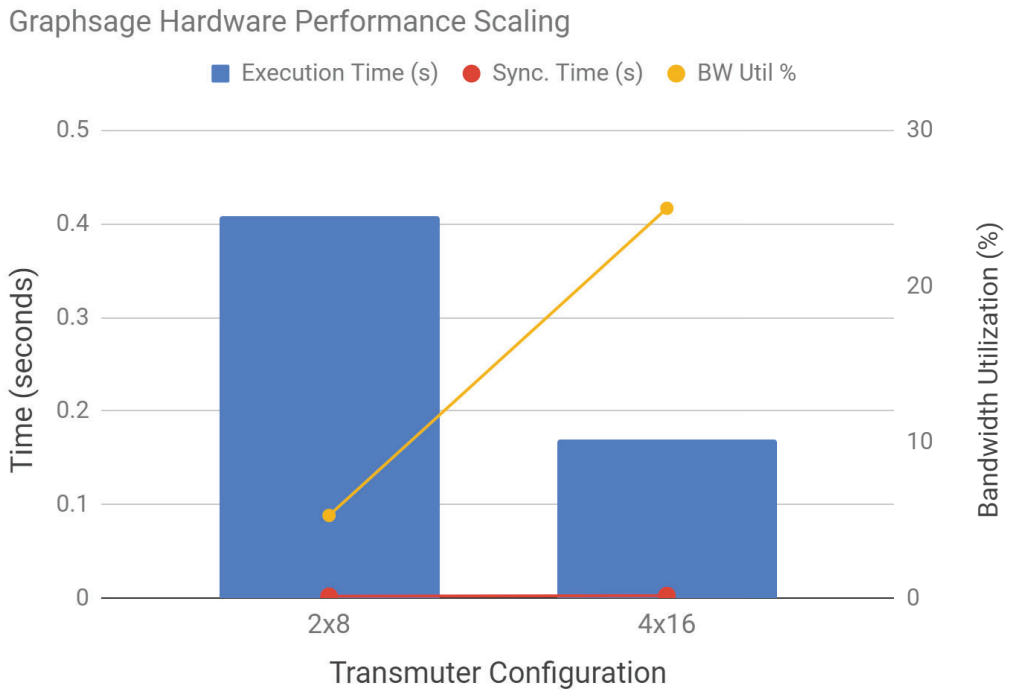


Figure 86: GraphSAGE Hardware Performance Scaling

5.2 IPNSW Analysis – Phase 1

Overview: IPNSW (Inner Product Navigable Small World), is a "hierarchical similarity graph"-based retrieval algorithm for approximate maximum inner product search (MIPS), and the task is only implement the query portion of algorithm.

5.2.1 Architecture Overview

In general, IPNSW has two parts of the algorithm:

- **Hierarchical greedy walk**: Walk through the given graphs and find the possible candidate clusters to start beam search. It takes around 10% of overall runtime, and major kernels contains dot product and graph traTransmuter v1.0l.
- **Beam Search**: Start from the candidate cluster, explore the graphs more thoroughly and use heaps to record the possible results. It takes around 90% of overall runtime, and major kernels contains dot product and heap algorithm (insert and pop). Each query should maintain its own heaps.

Because searching operations of each query are independent, there are **512** queries in the workload, queries are partitioned to each tile evenly and scheduled to available GPE dynamically based on the assumption that the workload per query can be viewed as uniform distribution. Most operations are fixed-point, floating-point operations only take around 10%, which are related to dot product.

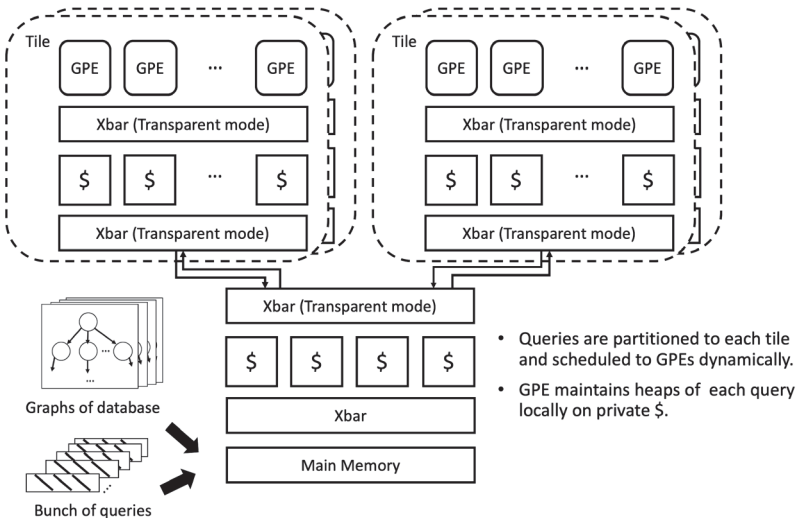


Figure 87: Workload Mapping on 4x16 Transmuter

Transmuter default configuration (4x16) is 4 tiles, 16 GPEs/tile, 4 KB L1 cache and 64 KB L2 cache. Both L1 and L2 work at shared cache mode in default. Reconfiguration mode of Transmuter is not used in this workload because the second step (Beam Search) takes the major part of execution time.

5.2.2 Performance Results

Comparisons of several metrics such as GOPs, GOPs/W and L1/L2 hit rate with different L2 cache size setup (4KB, 16KB, 64KB) and operation mode (shared cache vs private cache) are shown in this part. Implementation v0 and v1 are also discussed here. The major difference between the two is the way to query the index of the node. In v0, indices of queries are stored in an array and the complexity of querying the index is O(n). In v1, a lookup table is added to optimize this part and the complexity becomes O(1). Because the great number of nodes (1 million), the number of access to L1/L2 cache is reduced to ~5% / ~3.5%, which results in ~16x and ~17x improvement on 2x8 Transmuter in terms of throughput and energy efficiency. Power is estimated with 14 nm technology node and the clock frequency of the whole system is 1 Ghz. Because of limited time, two implementations are only compared in small system (2x8 Transmuter). The experiments of 4x16 Transmuter are also put at the end of this part.

The initialization involves loading >1GB of data and synchronizing so that all GPEs can start at the same time. This is done in parallel by the four LCPs and costs 0.317 sec and 0.0147W of for DARPA's workload (512 queries) on 4x16 Transmuter of 64 KB L2 cache bank size. It's important to notice that other GPEs and LCPs start working after initialization, and thus the number of initialization should be the same among all configurations of the same L2 cache size. The number shown in this section is only for computation part to focus on the computing performance of different configurations and implementation. That is, all statistics in this section do not include the initialization stage.

5.2.2.1 Implementation v0:

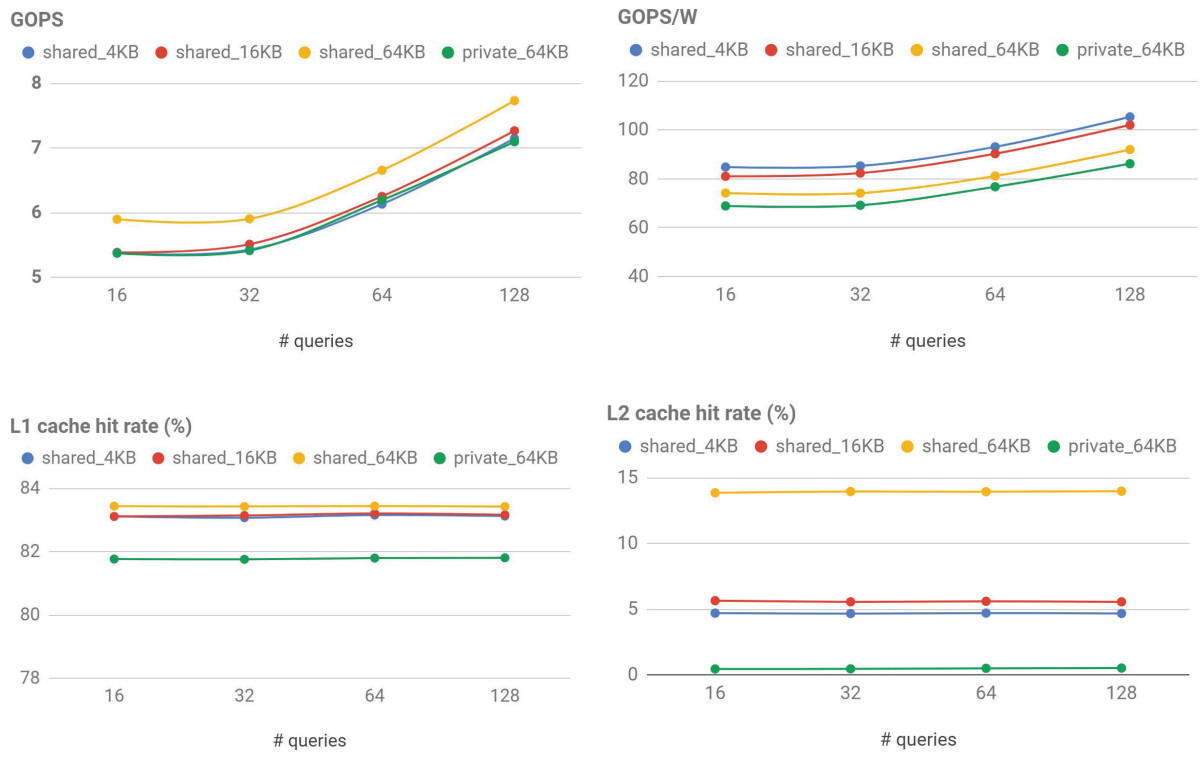


Figure 88: GOPs, GOPs/W, L1/L2 Cache Hit Rates of Implementation v0 on 2x8 Transmuter

There is a linear growth in GOPs and GOPs/W as the number of queries increases among all setups, while L1/L2 cache hit rate seems to be independent with the number of query. Transmuter in shared cache mode is better than private cache mode in terms of throughput and energy efficiency. Tranmuster in small L2 cache size gives better energy efficiency but worse throughput. L2 cache hit rate among those setups are generally bad, and the hit rate of private cache mode is the worst (< 1%).

In short, though, private cache mode is considered to be the best at first among other configurations due to many independent queries, the performance is actually the worst resulted from the significant memory read request of querying the index of node in implementation v0.

Notes: Because of the limited time, we scale down the workload and show #queries from 16 to 128 in v0.

5.2.2.2 Implementation v1:

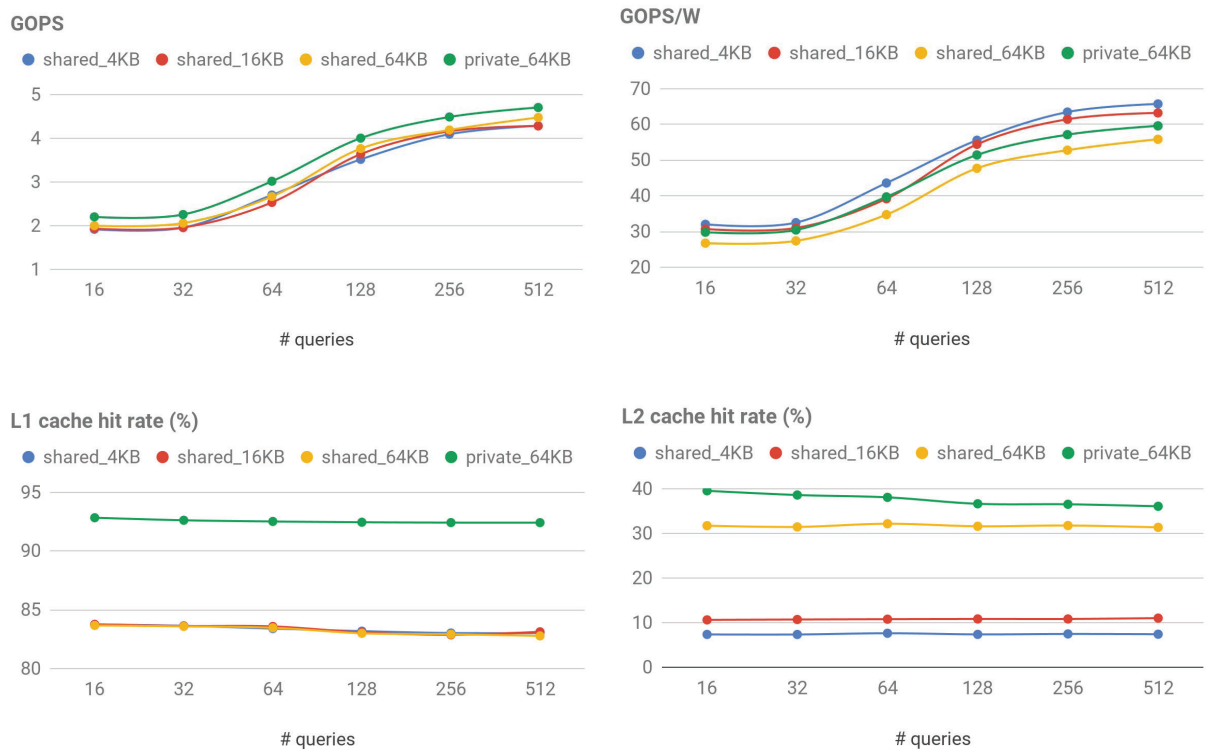


Figure 89: GOPs, GOPs/W, L1/L2 Cache Hit Rates of Implementation v1 on 2x8 Transmuter

In Figure 89, there is also a linear growth in GOPs and GOPs/W when the number of queries ranges from 32 to 128 and the growth gradually slows down when the number of queries keep increasing. L1/L2 cache rate in shared cache mode have little fluctuation among different number of queries, but a slight decrease in private cache mode is shown as the number of queries increases above 128.

Owing to the improvement on querying the index of node, lots of unnecessary memory read requests are removed, Transmuter in private cache mode is much better than shared cache mode in terms of throughput and cache hit rate. Besides, private cache mode is better in terms of energy efficiency compared with shared cache mode of the same L2 cache size. Fewer memory read requests gives better L1/L2 cache performance in general, though it makes throughput and energy efficiency "seemingly" worse than that in implementation v0.

5.2.2.3 Implementation v0 vs v1:

Performance and energy efficiency are both important metrics for final evaluation, and thus those metrics between implementation v0 and v1 are shown in Figure 90. Because of different implementation, there is an obvious gap in operations count and hence influence the number such as GOPs, we also put the overall runtime and energy delay product (EDP) for reference and the vertical axis are both in log scale. The default setup of transmuter is shared cache mode and the

private cache mode has the best QoR in implementation v1; therefore, these 3 combinations of implementation and implementation are compared in the below charts. All of them have 64 KB L2 cache.

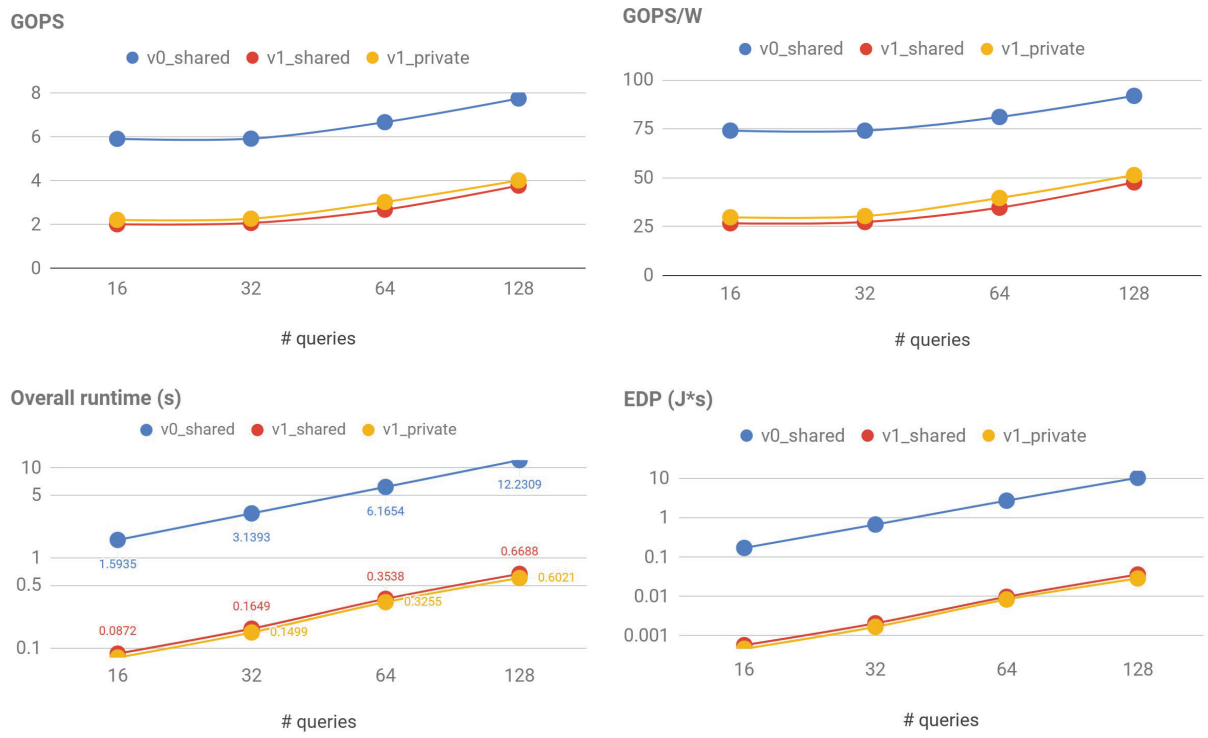


Figure 90: Comparison between Implementation v0 and v1

In conclusion, though GOPs and GOPs/W in v1 are lower than that in v0, v1 actually has much less execution time and energy consumption.

5.2.2.4 Implementation v1 on default system (4x16 Transmuter):

Ideally, when the Transmuter is scaled up from 2x8 to 4x16, it is expected to see x4 speedup in terms of throughput since there are 4 times more GPEs (64 vs 16) in the system. However, larger system also implies that potential decline on the cache hit rate. L2 cache hit rate (Figure 91) is influenced the most, which is only 50% of that in 2x8 Transmuter. Though increasing L2 cache miss rate degrade the ideal speedup on performance, there is still a 1.8x improvement on throughput. Besides, private cache mode is still more desired in 4x16 Transmuter because shared cache mode suffers from more serious deterioration of L2 cache hit rate.

However, power consumption is always the Achilles' heels in larger system such as 4x16 Transmuter. In our estimation, averaged power increases 2.4x compared with 2x8 Transmuter. GOPs/W is lower in 4x16 system because higher power consumption destroys the benefits from throughput improvement. Fortunately, power consumption can be improved by standard engineering techniques such as clock gating, which would be addressed later.

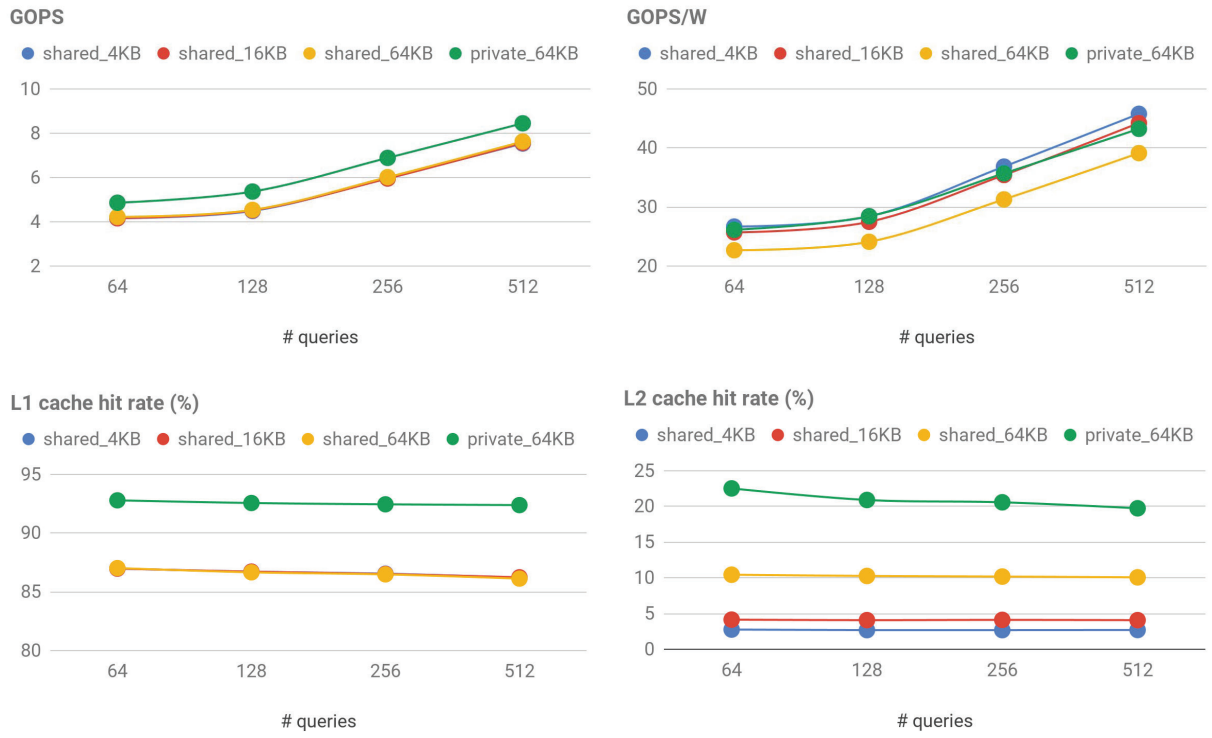


Figure 91: GOPs, GOPs/W, L1/L2 Cache Hit Rates of Implementation v1 on 4x16 Transmuter

5.2.2.5 2x8 Transmuter vs 4x16 Transmuter:

To have more insights into the larger system, several metrics such as GOPs, GOPs/W, overall runtime, and EDP of 2x8 and 4x16 Tranamuter with both shared cache mode and private cache mode are compared in Figure 92. All configurations have 64KB L2 cache bank. Growth of GOPs and GOPs/W on 2x8 Transmuter becomes much slower when the number of queries is greater than 256; while there is still a linear growth in 4x16 system. Similar linear growth can be observed in 2x8 Transmuter when the number of queries ranges from 64 to 128. The key factor of the linear growth is the averaged number of queries on each GPE per tile.

For example, the averaged number of queries on each GPE per tile is 4 and 8 in 2x8 Transmuter with 64 queries and 128 queries in total, respectively; the averaged number of queries is 4 and 8 in 4x16 system with 256 queries and 512 queries in total. In other words, performance measurement of different Transmuter's configuration should be based on the same averaged number of queries on each GPE and Transmuter can be expected to reach the peak performance when averaged number of queries on GPE is 16 or 32. Besides, a poor GOPs improvement can be seen on both 2x8 and 4x16 Transmuter when averaged number of queries on GPE is changing from 1 to 2. It is an important observation to do the projection of performance on much larger system such as 64x64 Transmuter, which would be addressed in the section of Extrapolation Performance on Larger System, and the reference to apply gating technique on those inefficient GPEs.

In short, when there are lots of queries running on the system, larger configuration such as 4x16 Transmuter is preferred to get the high performance. Though power consumption is higher on

4x16 configuration, total energy consumption (see Table 11 and Additional Datapoints Section) is actually similar due to the less execution time. Therefore, 4x16 Transmuter is more energy-efficient in terms of EDP.

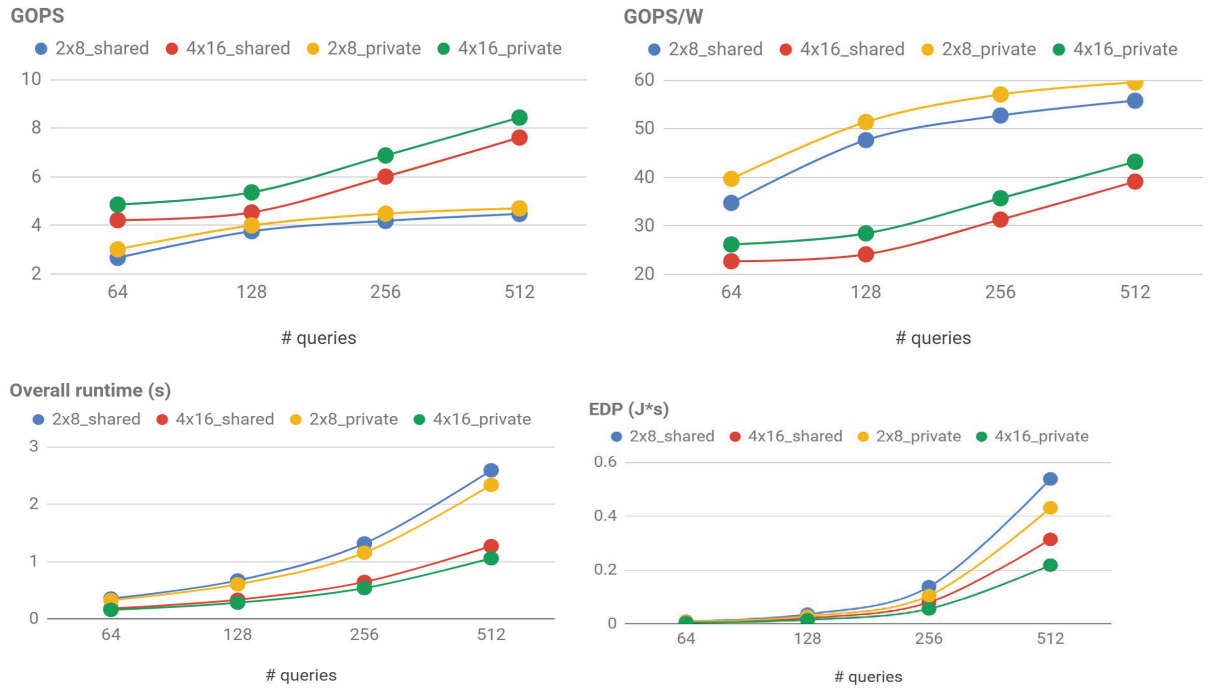


Figure 92: Comparison between 2x8 Transmuter and 4x16 Transmuter

5.2.3 IPNSW Performer Performance Tables

The best performance of Transmuter configuration in this workload is private cache mode with implementation v1 and we put the statistics of the best configuration in Table 16. To show the difference directly, statistics of default configuration is shown in Table 15. Both configurations have 64KB L2 cache bank.

Table 15: Performance Table of Default Configuration (shared cache mode) in 4x16 Transmuter

# queries	GOPs/Watt	Execution time (s)	Total energy (J)	Simulated/ estimated cycles	L1/L2 Cache hit rates (%)	Percentage of system simulated
64	22.6773	0.1790	0.0333	178999050	87.02 / 10.43	100%
128	24.1089	0.3325	0.0625	332536168	86.66 / 10.26	100%
256	31.2848	0.6431	0.1236	643148624	86.48 / 10.17	100%
512	39.1178	1.2691	0.2472	1269078004	86.12 / 10.06	100%

Table 16: Performance Table of Default Configuration (private cache mode) in 4x16 Transmuter

# queries	GOPs/Watt	Execution time (s)	Total energy (J)	Simulated/ estimated cycles	L1/L2 Cache hit rates (%)	Percentage of system simulated
64	4.8576	0.1549	0.0288	154944306	92.78 / 22.51	100%
128	5.3605	0.2810	0.0530	280993944	92.55 / 20.89	100%
256	6.8818	0.5351	0.1032	535146628	92.44 / 20.57	100%
512	8.4424	1.0563	0.2063	1056348043	92.38 / 19.73	100%

5.2.4 Scalability

Projections on the performance if the data size becomes larger: In the implementation of this workload, each GPE processes queries dynamically and the scheduling is scalable. We can claim that Transmuter can work for any data size in this workload unless the DRAM cannot fit the input data. As the metrics in the Performance Table, both execution time and energy consumption have a linear increase as the number of queries grows, while the growth of GOPs/W gradually slow down when averaged number of queries on GPE is larger than 32. Therefore, it is expected to see a linear increase of execution time when Transmuter runs on the larger dataset, e.g. more queries such as 1024.

5.2.5 Extrapolation Performance on Larger System

In the future, we plan to scale our design up to 64x64. Hence, an experiment of given workload with implementation v1 on different configuration of Transmuter (2x8, 4x16, 8x16) is conducted to see the performance gain. All configurations have 64KB L2 cache and L1/L2 cache are configured as shared cache mode.

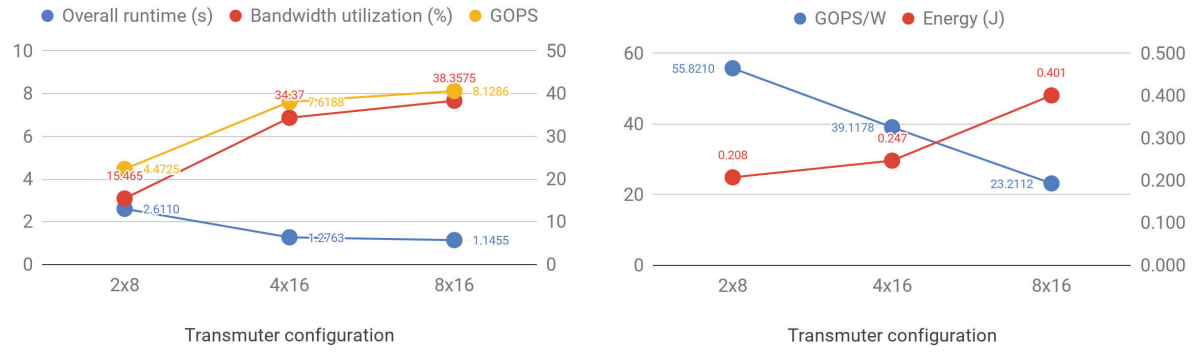


Figure 93: Total Execution Time, Bandwidth Utilization, GOPs/W, and Energy Across Different Configurations

A linear speedup in terms of the execution time is expected and memory bandwidth might be the limitation of performance, but the experiment results do not meet the previous expectations. We conclude that it is because of the averaged number of queries on GPE and the non-uniform distribution of the workload per query (Figure 93). We use the number of dot products per query as the index of workload per query. The number of dot products per query on average is 3385, median is 3319, minimum is 1991, and maximum is 5659.

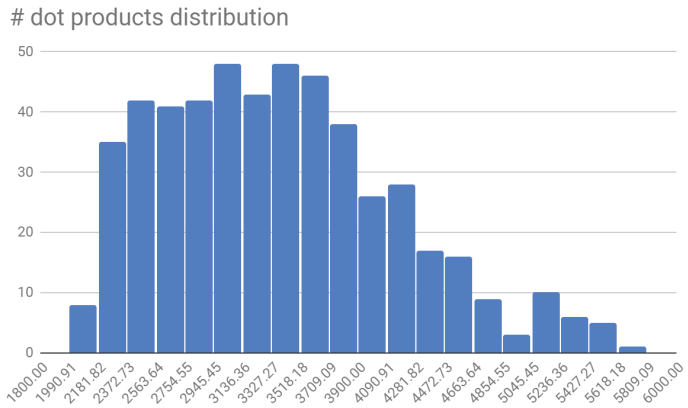


Figure 94: Distribution of the Number of Dot Products per Query
There are 512 queries in total.

Apparently, the workload per query is not uniform, which destroyed the assumption of scheduling policy. When each GPE on tile is assigned "enough" queries to run, it can somehow hide the latency caused by the non-uniform workload per query; otherwise, the performance gain would be limited to the query with the heaviest workload. In 2x8 system, each GPE/tile can run 64 queries on average; in 4x16 system, each GPE/tile can run 8 queries on average, and in 8x16 system, each GPE/tile can only run on 4 queries on average. Hence, we propose a solution in the section of future work to optimize the scheduler to balance the workload across tiles.

In addition, when averaged number of queries on GPEs/tile is not enough, energy efficiency (GOPs/W) and performance (GOPs) are also degraded according to the observation from section of Performance Results. In brief, we can expect a linear growth on both execution time and memory bandwidth on larger configuration of Transmuter if there are "enough" queries (at least 16 queries on average) for each GPE to run, though energy efficiency is degraded because of the larger system's overhead .

To have a fair projection of performance and energy efficiency, a normalized comparison is necessary by running simulations on different configurations (2x8, 4x16) with 8 queries running on GPEs/tile. That is, there are 128 queries running on 2x8 Transmuter and 512 queries on 4x16 Transmuter.

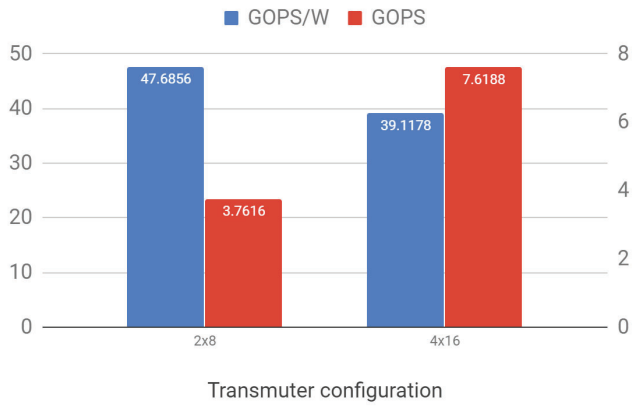


Figure 95: Normalized Comparison between Different Transmuter Configurations

We can see there are 2x improvement on performance while 18% energy efficiency loss which can be optimized by standard engineering techniques such as clock gating, which are going to be addressed in the next section. Moreover, we can shut off some GPEs when the averaged number of queries is not enough (less than 8) to maintain a good performance.

In conclusion, if there are enough queries running on GPEs and we can do dynamic work allocation across all GPEs with an improved scheduler, we can expect that memory bandwidth utilization will hit around 100% on 32x32 Transmuter, and the peak of GOPs is limited by the memory bandwidth. Hence, in 64x64 system, the peak performance is roughly **36.57 GOPs** and, with the help of standard engineering techniques, peak energy efficiency is roughly **60.59 GOPs/Watt .**

5.2.6 Estimated Improvements by Standard Engineering Techniques

The improvement of these techniques depends on the workload as well as the data size. We give a rough estimation here.

- **Clock gating**: It can only reduce the dynamic power. Though the effect depends on the switching activity of clocked component, it generally can give **30%-50%** total power reduction with **20%-30%** area increase overhead from previous experiences.
- **Power gating**: It is only applicable to use during idle state to have the energy reduction. In IPNSW, the performance gain is poor when averaged number of queries is only 1 or 2, and hence we can shut off some GPEs when the averaged number of queries is less than 2 to reduce energy consumption. In general cases, this technique can reduce total power to **2% or less** compared to active power dissipation with **less than 10%** area increase.
    - Power Gating and Dynamic Leakage Optimization for SRAM: In general, it can reduce power 20%-40% in "retention mode" compared to active mode of cache. The actual effect depends on utilization rate of cache. In IPNSW, it is estimated to reduce around 2.7% power in total.
- **Near-threshold operation**: It is an optional choice because it can reduce power consumption at the cost of performance reduction. From previous experience, it can reduce total power to 4x-8x depending on technology node, but it also reduces performance by 6x-10x.
- **Parallel barrier synchronization**: Synchronization time increases when there are more GPEs in the system. Actual improvement of the technique depends on the workload and implementations.

Again, an experiment of synchronization time is conducted among different configurations of Transmuter (2x8, 4x16, 8x16) with the same workload of 512 queries in total. The result is shown in Figure 96 and the percentage of the overall runtime is also shown here.

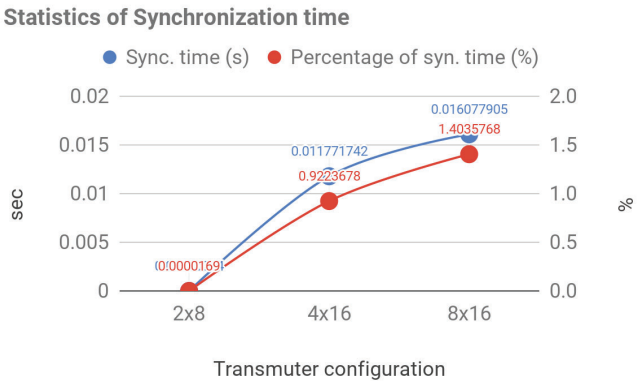


Figure 96: Statistics of Synchronization time in IPNSW

Though the percentage of synchronization time increase as the number of GPEs increase, it only takes around 1 % on 8x16 Transmuter. However, parallel barrier synchronization should have some benefits in a larger system such as 64x64 Transmuter in terms of performance.

5.2.7 Additional Datapoints

We provide the default and best performance table of 2x8 Transmuter for reference. Default configuration works in shared cache mode, and the best configuration works in private-cache mode. Both configurations have 64 KB L2 cache.

Table 17: Performance Table of default configuration (shared cache mode) in 2x8 Transmuter

# queries	GOPs/Watt	Execution time (s)	Total energy (J)	Simulated/ estimated cycles	L1/L2 Cache hit rates (%)	Percentage of system simulated
16	26.745	0.0872	0.0065	87226043	83.67 / 31.69	100%
32	27.381	0.1649	0.0124	164945946	83.59 / 31.42	100%
64	34.728	0.3538	0.0272	353770706	83.47 / 32.13	100%
128	47.686	0.6688	0.0528	668751941	83.00 / 31.56	100%
256	52.769	1.3155	0.1044	1315536987	82.92 / 31.74	100%
512	55.821	2.5930	0.2078	2592987059	82.78 / 31.32	100%

Table 18: Performance Table of Default Configuration (private cache mode) in 2x8 Transmuter

# queries	GOPs/Watt	Execution time (s)	Total energy (J)	Simulated/ estimated cycles	L1/L2 Cache hit rates (%)	Percentage of system simulated
16	29.812	0.0790	0.0058	79030292	92.83 / 39.51	100%
32	30.433	0.1499	0.0111	149922905	92.62 / 38.56	100%
64	39.695	0.3255	0.0253	325457352	92.51 / 38.04	100%
128	51.429	0.6021	0.0468	602084100	92.45 / 36.60	100%
256	57.114	1.1534	0.0906	1153388903	92.42 / 36.49	100%
512	59.580	2.3370	0.1844	2336957732	92.41 / 36.04	100%

When using private cache mode, we can see a 10% improvement in terms of execution time and GOPs/Watt. Besides, we can also see a 5% to 10% increase on L1/L2 cache hit rates in private-cache mode.

5.3 Recsys Analysis – Phase 1

This task implements and trains a neural network based recommender system. The neural network takes in a list of movies that a user has liked and tries to predict a score for all other movies. As it is an autoencoder-based system, the input is first encoded into a hidden representation (hidden_dim), which has a dimension less than the input dimension (emb_dim), and then decoded back into a higher dimension (output_dim). This is will help the neural network to learn something new rather than just learning an identity mapping. The architecture for the neural network is shown in Figure 97.

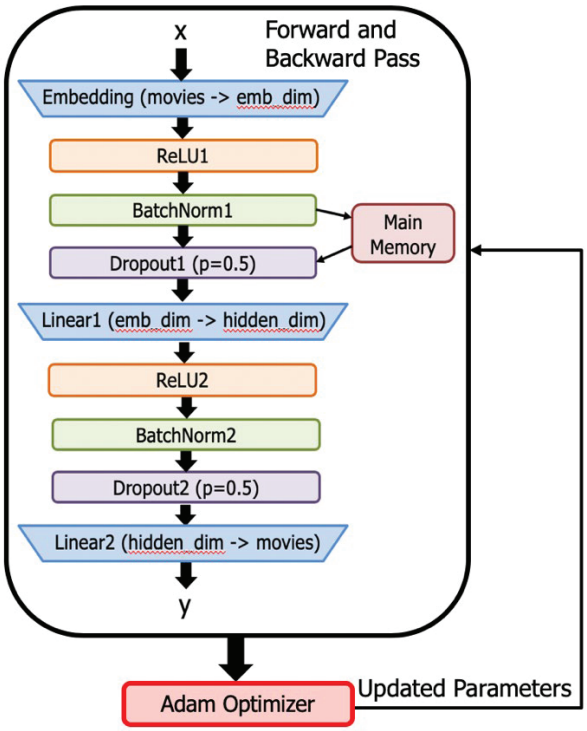


Figure 97: Neural Network for Recommender System

The network consists of the embedding layer, ReLU, 1-D batch normalization, dropout and fully connected layers. The user information is first embedded into embedding_dimension, then to hidden_dimension and finally to output_dimension which is the number of movies. During the training phase, a group of users form a batch and after each batch's forward and backward pass, the weights are updated using the Adam Optimizer. The output is sent through a sigmoid layer to measure the binary cross entropy loss.

5.3.1 Configuration/Dataset

The memory configuration used is default i.e., shared cache for both L1(4 Kb) and L2(64 kb) memory banks with 4 tiles and 16 GPEs each. The DARPA dataset has 69878 users and 10678 movies.
The hyper parameters of the network are as follows,

- batch_size: 256; embedding dimension: 800; hidden dimension: 400; output dimension: 10678;
- bias_offset: -10; dropout probability: 0.5
- Optimizer: ADAM optimizer; learning rate: 0.01; betas: (0.9, 0.999); weight decay: 0; epsilon: 1e-8
- loss: binary cross entropy

5.3.2 Mapping on Transmuter

The mapping of all the kernels on the Transmuter is discussed below. As the configuration is shared cache, all the weights and results are stored in the main memory. For this reason, the results after each layer are stored and fetched from the main memory as shown in Figure 97.

Embedding Layer: The embedding layer has a weight matrix called the embedding matrix that has rows equal to the number of movies and columns equal to the embedding dimension. The values of the matrix are initialized randomly using *Normal(0,0.01)*. In the forward pass, the input to the embedding layer is the list of movies a user has rated, and the output is the sum of all the 800-dimensional vectors that resemble the movies. Each GPE computes the 800-dimensional vector of each user. A total of 4 users are assigned to each GPE.

Similarly, for the backward pass, the gradient of a row of the embedding matrix is the sum of the 800-dimensional vectors (of the incoming gradients) if a user has rated that movie. This can also be mapped by assigning each movie (row) to each GPE.

ReLU: In the forward pass, ReLU adds a non-linearity by passing only the values which are greater than zero, rest all are assigned zeros. As it is an element wise operation, the dimensions are unchanged. The backward pass is computed by multiplying the incoming gradient with the ReLU mask (comprised of 1's and 0's based on the values passed initially) which is then sent to the previous layer. Both the forward and backward pass are mapped by assigning each user's vector to each GPE (4 users each).

1D Batch Normalization: This layer has two weight vectors called *gamma* and *beta*. The length of the vectors is same as the dimension from the previous layer. The first batch normalization layer has the size 800 and the second one has the size 400. The *gammas* are initialized with 1's and the *betas* with 0's. The calculation of mean, variance, normalized values and the final output of each dimension are mapped to each GPE.

The backward pass comprises of three gradient calculations that include the input gradient, gamma gradient and beta gradient. The input gradient is calculated by assigning each user to each GPE. The gamma and beta gradients are calculated by assigning each dimension to each GPE.

Dropout: In the forward pass, each element is either passed or dropped with a dropout probability of 50%. The passed values are scaled by 1-p, where p=0.5. Similar to ReLU, a dropout mask (comprised of 1's and 0's) is generated which is used in backward pass to multiply with the incoming gradients. Both the forward and backward pass are mapped by assigning each user's vector to each GPE (4 users each).

Fully Connected Layer: The weight matrices of the hidden layer and the output layer are initialized randomly using *Uniform(min=-scale, max=scale)* where *scale* is *1/emb_dim* and *1/hidden_dim* respectively, their sizes are *(hidden_dim x emb_dim)* and *(output_dim x hidden_dim)* which are initially stored in transposed form. The bias vector of the hidden layer is initialized with zeros and that of the output layer with -10 (bias_offset).

The forward pass is a matrix multiplication of the previous layer's output with the transposed weight matrix followed by bias addition. The matrix multiplication is divided into blocks of size 16 and is assigned to each GPE. The backward pass comprises of three gradient calculations that include the input gradient, weight gradient and bias gradient. For the matrix dimensions to match, two transposes are performed (each row assigned to each GPE) which are the incoming gradient and the weight matrix. The input gradient is a matrix multiplication of the incoming gradient with the weight matrix. The weight matrix gradient is a matrix multiplication of the transposed weight gradient with its input (forward pass). Both the matrix multiplications are mapped as in forward pass. The bias gradient is the addition of incoming gradient across its dimensions where each dimension is mapped to each GPE.

Binary Cross Entropy Loss: Each GPE computes the loss for each user. After all the losses are computed, LCP0 averages them to get the final loss. The loss computed for the first batch is 0.102.

Adam Optimizer: The parameters that are to be updated are the embedding matrix, gammas, betas, weight matrices and biases. The updation of matrices are mapped by assigning each row to each GPE and the updation of the vectors by assigning each dimension to each GPE.

5.3.3 Performance Results on DARPA Dataset

The initialization part is done by one of the LCP and takes about 1.25s. It includes

1. Loading the DARPA dataset, which is stored in Numpy array file (.npy), onto the DRAM.
2. Initializing the weight matrices for the Embedding layer (embedding matrix), BatchNorm (gammas and betas), Linear layer (Weights and biases) and Adam Optimizer.

Table 19 shows the overall results for a batch of 256 users on DARPA dataset. Since one batch takes around 2 days to run in Gem5, we could not run this algorithm for 69878 users. However, the data for 69878 users could be processed in 69878/256 ≈ 273 batches. Since the embedding layer (both forward and backward) would slightly vary for each batch and the final batch would have only 246 users, we can project that the execution time for the entire dataset would be close to 273 times the execution time of one batch of 256 users, i.e. 289.926s. The initialization cost is thus distributed across all 273 batches.

The breakdown of the execution time for one batch is shown in Figure 98. The list of metrics for the forward pass is shown in
Table 20 and for the backward pass (with Adam Optimizer) is shown in Table 21.

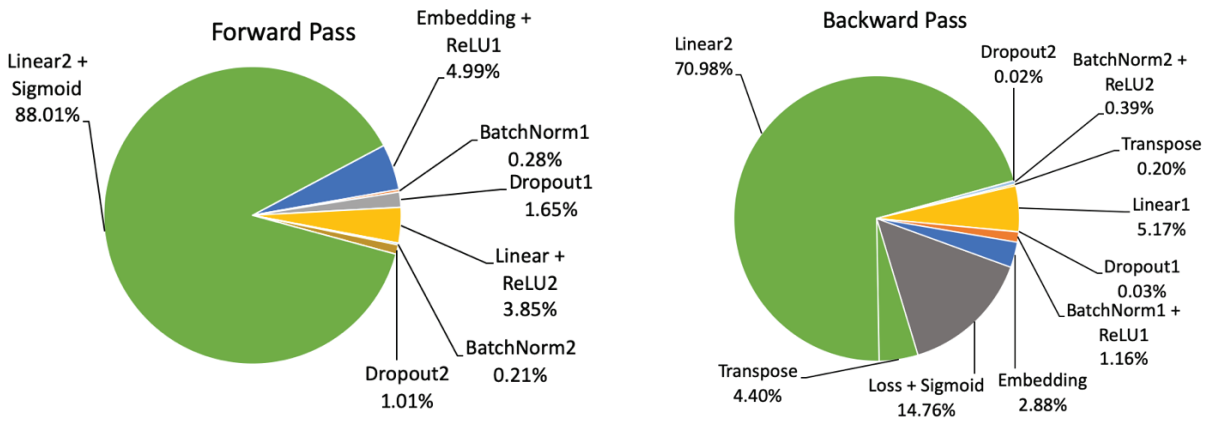


Figure 98: Breakdown of Execution Time for Forward (left) and Backward (right) Pass for Batch Size of 256

Table 19: Performance Data for 256 users on DARPA Dataset

Execution Time (s)	GOPS	GFLOPS	GOPS/W	GFLOPS/W	Total Energy (J)	Simulated/ Estimated cycles	Cache hit rates in %(L1,L2)	%Sys.Simulated
1.062	39.02	12.59	169.12	54.55	0.245	1062M	98.45, 22.05	0.4%

Table 20: Breakdown of Forward Pass (35.1%)

Layer	Time (ms)	GOPS	GFLOPS	GOPS/W	GFLOPS/W	Total Energy (J)	Cache hit rates in % (L1, L2)
Embedding+ReLU1	18.62	10.36	4.86	53.70	25.21	3.59 E-03	96.82, 9.19
xBatchNorm1	1.03	28.51	3.93	130.45	18.00	2.20 E-04	95.28, 38.66
Dropout1	6.14	37.36	0.36	165.57	1.62	1.38 E-03	96.00, 56.61
Linear1 + ReLU2	14.36	53.77	18.57	186.35	64.37	4.14 E-03	99.00, 18.42
BatchNorm2	0.77	19.44	2.64	94.75	12.86	1.50 E-04	95.04, 29.20
Dropout2	3.77	30.42	0.29	140.79	1.38	8.10 E-04	95.97, 57.81
Linear2 + Sigmoid	328.11	38.77	11.43	150.75	44.45	8.43 E-02	98.41, 27.77

Table 21: Breakdown of Backward Pass (47.7%) and Adam Optimizer (17.2%)

Layer	Time (ms)	GOPS	GFLOPS	GOPS/W	GFLOPS/W	Total Energy (J)	Cache hit rates in % (L1, L2)
Loss + Sigmoid	74.78	22.99	8.45	105.91	38.92	1.62 E-02	94.02, 26.76
Transpose2	22.28	38.72	1.7 E-05	178.36	8 E-06	4.83 E-03	67.33, 5.16
Linear2	359.56	55.68	19.79	193.50	68.76	0.103	99.17, 14.30
Dropout2	0.08	12.25	6.02	61.85	0.40	1.68 E-05	91.35, 0
BatchNorm2+ ReLU2	1.99	42.56	4.01	182.83	17.24	4.6 E-04	93.03, 25.28
Transpose1	1.02	47.74	6.2 E-05	211.55	2.7 E-04	2.2 E-04	71.72, 12.20
Linear1	26.19	57.30	20.33	201.65	71.54	7.44 E-03	99.18, 11.58
Dropout1	0.165	12.51	6.20	62.85	31.17	3.29 E-05	91.52, 0.01
BatchNorm1+ ReLU1	5.86	29.87	2.73	137.58	12.60	1.27 E-03	92.56, 23.70
Embedding	14.58	35.71	15.32	135.19	58.00	3.85 E-03	99.41, 14.86
Adam Optimizer	182.69	13.28	3.95	64.61	19.21	3.75 E-02	97.03, 31.07

5.3.4 Extrapolation Performance on Larger Dataset

- Increasing the number of users doesn't affect the computations per batch because each mini-batch has a fixed set of users and is independent of the other. A slight impact will be on the embedding layer if the number of ratings by each user increases or decreases because the embedding layer sums up all the movies' vectors from the embedding matrix.
- Increasing the number of users will have a linear increase of the execution time due to the increase in number of batches.
- Increasing the number of movies impacts the output layer i.e., a linear increase in the execution time of the output layer.
- If the sizes of other parameters (emb_dim, hidden_dim) increase, the major impact would be to the fully connected layer as it increases the number of multiplications. It would also affect the transpose. All other layers except for the batch norm have element-wise operations, the change in their execution time wouldn't be that significant. For the batch norm, the dimensions for calculating the mean and variances would increase.

5.3.5 Extrapolation of Performance on Larger System

Increasing the number of tiles and number of GPEs per tile will have a positive effect on the overall execution time. We run the recommender systems on a scaled dataset using the three configurations. The hyperparameters of the scaled dataset are batch_size:128; emb_dim:512; hidden_dim:128 and output_dim:2048. For each configuration, we run one batch.
The three configurations are:

(1) **2x8**: 2 tiles and 8 GPEs per tile;
(2) **4x16**: 4 tiles and 16 GPEs per tile;
(3) **8x16**: 8 tiles and 16 GPEs per tile.

We test the execution time, synchronization time, bandwidth, and the energy for each execution. Table 22 presents the performance generated using different configurations

Table 22: Synchronization Performance for Recsys

Configuration	Execution Time (s)	Synchronization Time (s)	Bandwidth Util. (%)	Energy (J)
2 x 8	0.368847	4.33E-04	5.43	0.029
4 x 16	0.116344	6.84E-04	14.32	0.024
8 x 16	0.102595	0.0189	18.10	0.037

Figure 99 shows the trend for execution time, synchronization time, and bandwidth utilization based on the data executed using three configurations.

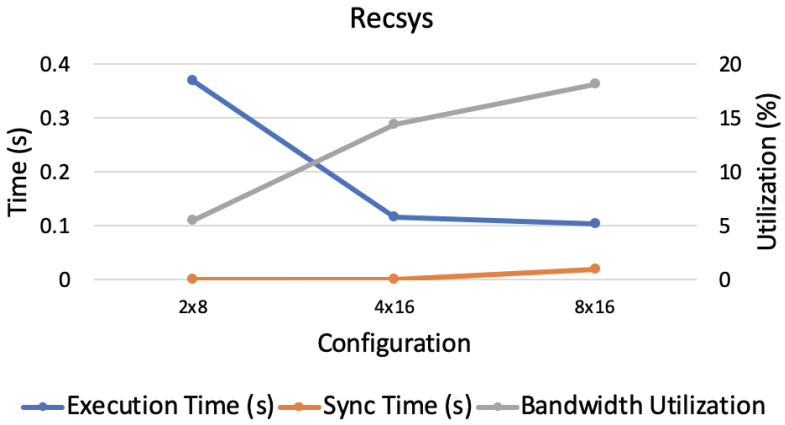


Figure 99: Trend of Execution time, Synchronization Time, and Bandwidth Utilization of Recsys

As the numbers of GPEs increase, the execution time reduces as expected. However, the execution time does not scale proportional as the number of GPEs increase. In our implementation, each user is assigned to each GPE and if the batch size is less than the number of GPEs, we can shut down the respective tiles/GPEs to save energy.

The bandwidth utilization and the synchronization time also increase as more GPEs are used for computation. On the other hand, decreasing the L2 cache size wouldn't help much as the previous layer's result is always fetched from the memory to L2 for the next layer's computation.

5.4 sinkhorn Analysis – Phase 1

The Sinkhorn algorithm determines similarity of various documents as an OT problem. The Sinkhorn algorithm is shown in Algorithm 1 below. In the Sinkhorn algorithm, the query document (query) and database (data) is vectorized using pre-trained word embedding and a bag of

words (BoW). The cost matrix (M) is derived from these pre-trained word embeddings. The BoW vectors tend to be very large and sparse, whereas M is dense. The inner-loop consists of two major kernels: a Masked dense-dense Matrix Multiplication (Masked-GeMM) and Dense-Sparse Matrix multiplication (DMSpM).

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

Algorithm 1. Sinkhorn Distance (MATLAB syntax)

5.4.1 Algorithm Mapping

The Masked-GeMM kernel is a variation of the GeMM kernel, with the addition of a sparse weight matrix. Rather than computing the full matrix multiplication, only the entries with non-zero elements in the weight matrix are computed. DMSpM is similar to the SpGeMM kernel, except that the first operand is a dense matrix, rather than a sparse matrix. To compute DMSpM, Transmuter uses a simplified outer product algorithm that splits the kernel into multiply (DMSpM-Multiply) and merge (DMSpM-Merge) phases. Figure 100 presents the mapping of the inner-loop kernels of the Sinkhorn algorithm onto Transmuter.

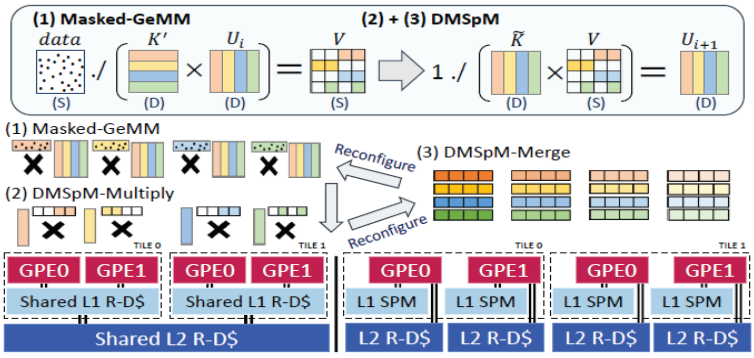


Figure 100: Mapping of the Sinkhorn algorithm on Transmuter

Sparse

(S) and Dense (D) indicate the nature of the input data. Sinkhorn iterates between Masked-GeMM and DMSpM. Transmuter is reconfigured before and after the merge phase of DMSpM for each iteration.

5.4.2 Results on DARPA Workload

Masked-GeMM and DMSpM-Multiply exhibit the best performance/shorter execution time in SMP (shared L1 cache + shared L2 cache) mode, due to good data reuse across GPEs. In DS (private scratchpad + private cache) mode, DMSpM-Merge shows 15.38% improved performance with increased power consumption. The detailed execution time and power efficiency is shown in Table 23, while the execution time breakdown is presented in Figure 101.

Table 23: Results on DARPA-Provided Datasets for Sinkhorn

Dataset	Config.	GOPS	GOPS/ W	Total execution time(s)	Simulated/ estimated cycles	Total energy (J)	L1/L2 hit rate (%)	%system simulated
Darpa	SMP	6.28	35.78	0.4452	445200000	0.07821	89.67 / 1.827	100%
	DS	4.7	27.37	0.7105	710500000	0.1240	N/A / 88.55	100%
	SMP+DS	6.13	34.80	0.4361	436100000	0.07689	89.63 / 29.50	100%

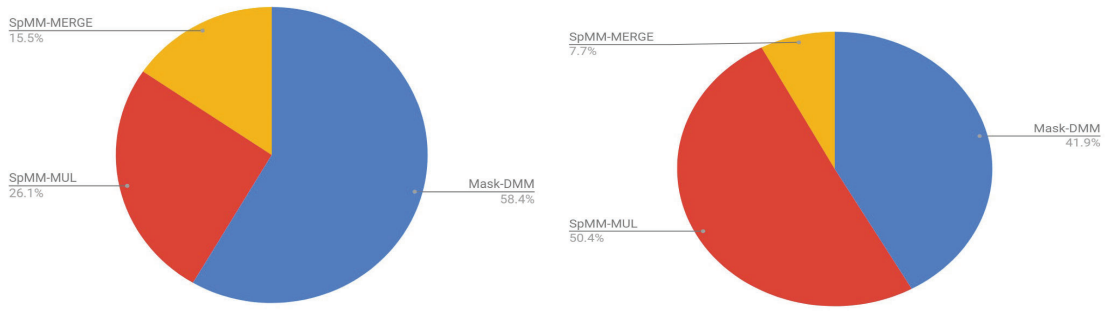


Figure 101: Execution Time Breakdown for (a) SMP Mode and (b) DS Mode

5.4.3 Results on other system configuration (2x8 and 8x16 Transmuter)

We also run Sinkhorn with DARPA datasets under different Transmuter size, and the results are reported in Table 24. Here the initialization is done by one LCP and costs 0.0679 sec. Increasing the size of Transmuter has a positive effect on reducing the overall execution time, as shown in Figure 102. The overall execution time keeps going down with larger Transmuter size.

Table 24: Results on different Transmuter size

Datasets	LCPSx GPES	Configuration	L1 hit rate	L2 hit rate	GOPS	GOPS/W	Total execution time(s)
Darpa datasets	2x8	SMP	86.42	29.42	3.52	48.97	0.72064
		DS	N/A	89.26	3.21	43.79	0.7841
		SMP+DS	87.89	46.55	3.45	47.87	0.6917
Darpa datasets	8x16	SMP	89.62	1.402	9.50	28.78	0.3939
		DS	N/A	89.10	7.54	22.94	0.6449
		SMP+DS	89.28	30.10	9.29	28.05	0.3864

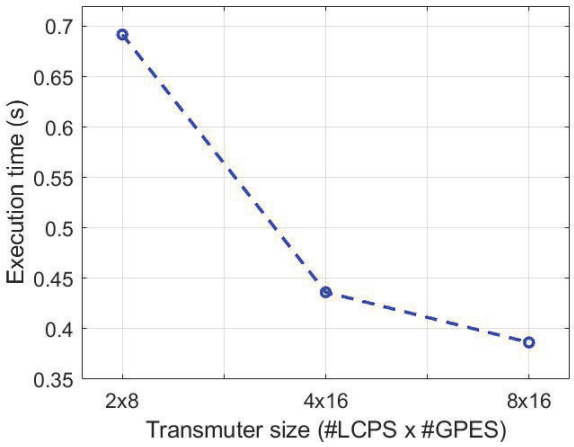


Figure 102: Total Execution Time Under Different Transmuter Size

5.4.4 More Experiments and Analysis

Previously, in the exploration of Sinkhorn with random-generated datasets, we find out that in each iteration of Sinkhorn the optimal power efficiency is gained by first performing the Mask-DMM and DMSpM-Multiply in SMP mode, later reconfiguring the Transmuter to DS mode for DMSpM-Merge. With DARPA dataset, the results show that runtime reconfiguration is not necessary for optimal efficiency. This indicates that even with the same workload, two datasets with different input sizes and densities may lead to a different reconfiguration strategy. In this case, DARPA data size is much larger (4x) and sparser compared to the random-generated datasets. To further motivate the need for reconfiguration, we scale down the size of DARPA dataset by an order of magnitude and increase the density of sparse matrix by 10x, 100x and 1000x. The results with three modified datasets are shown in Table 25 below.

Table 25: Results with Smaller Sized Datasets which has Lower Sparsity (higher density)

Datasets	Density	Configuration	L1 hit rate	L2 hit rate	GOPS	GOPS/W	Kernel execution time (s)
Modified Dataset 1	0.0034 (10X)	SMP	90.41	5.05	13.88	67.65	0.01540
		DS	N/A	89.56	11.69	54.56	0.01855
		SMP+DS	90.21	32.52	13.79	67.45	0.014837
Modified Dataset 2	0.034 (100X)	SMP	89.83	8.46	14.96	68.49	0.1394
		DS	N/A	88.66	10.94	52.52	0.1905
		SMP+DS	93.63	33.79	17.36	77.76	0.1079
Modified Dataset 3	0.34 (1000X)	SMP	88.57	2.93	15.05	66.51	1.387
		DS	N/A	87.64	10.43	50.52	1.999
		SMP+DS	92.23	28.20	17.25	74.69	1.092

From Table 25, we can observe that as matrix density increases, the benefit of reconfiguration also becomes larger. This is because with increased density, the SpMM-Merge phase takes more time to complete and becomes the bottlenecks of the application execution. And DS mode provides higher energy efficiency and performance over SMP mode. The results for three datasets with different density levels are visualized in Figure 103. Hybrid mode (SMP+DS) acheves reductions in both total runtime, energy consumption and energy-delay-product (EDP). Therefore, for these relative denser matrix, each iteration of Sinkhorn execution involves two reconfigurations: one at the beginning of DMSpM-Merge (SMP to DS) and another at the end of DMSpM-Merge to switch back to SMP for the next iteration.

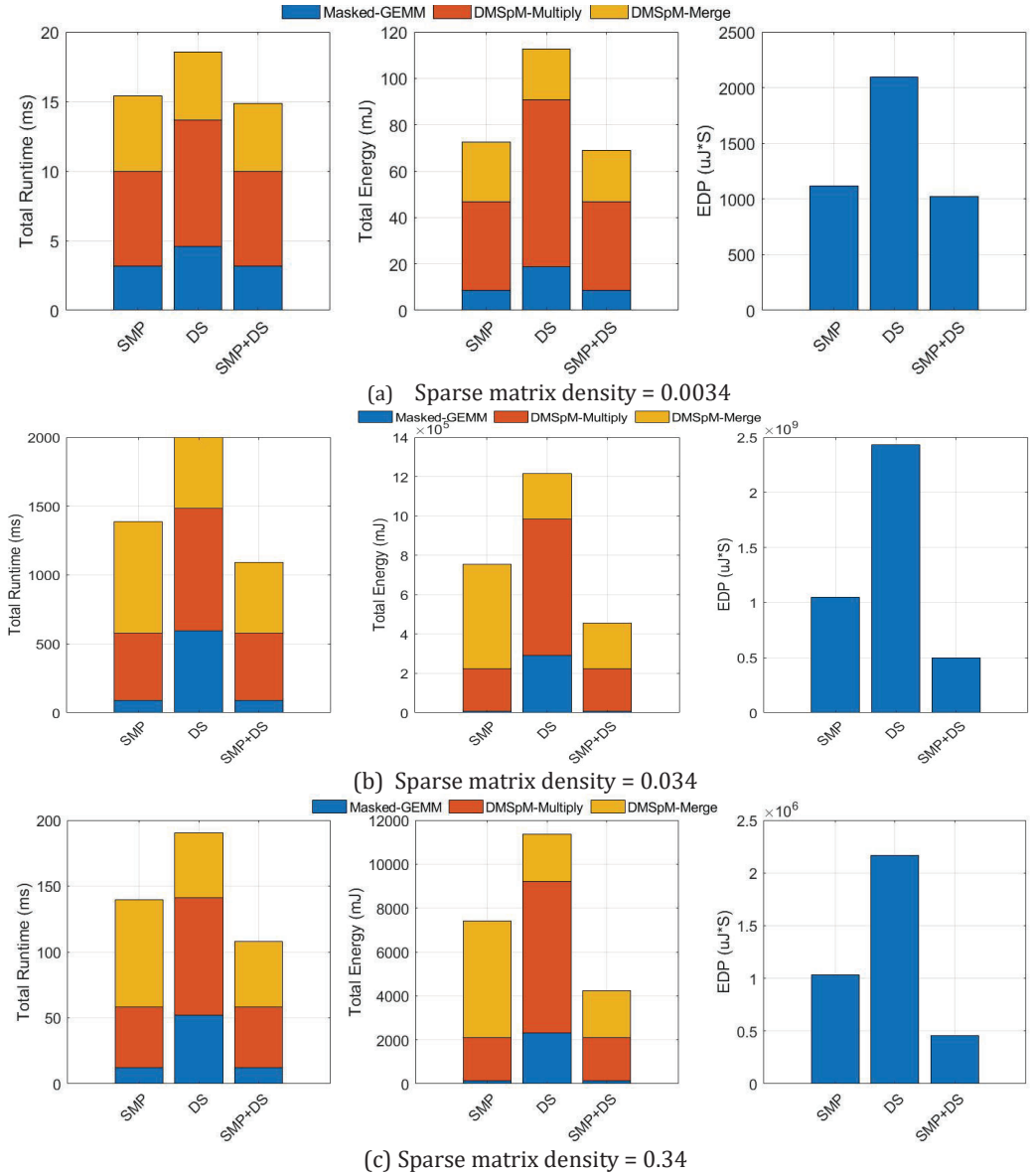
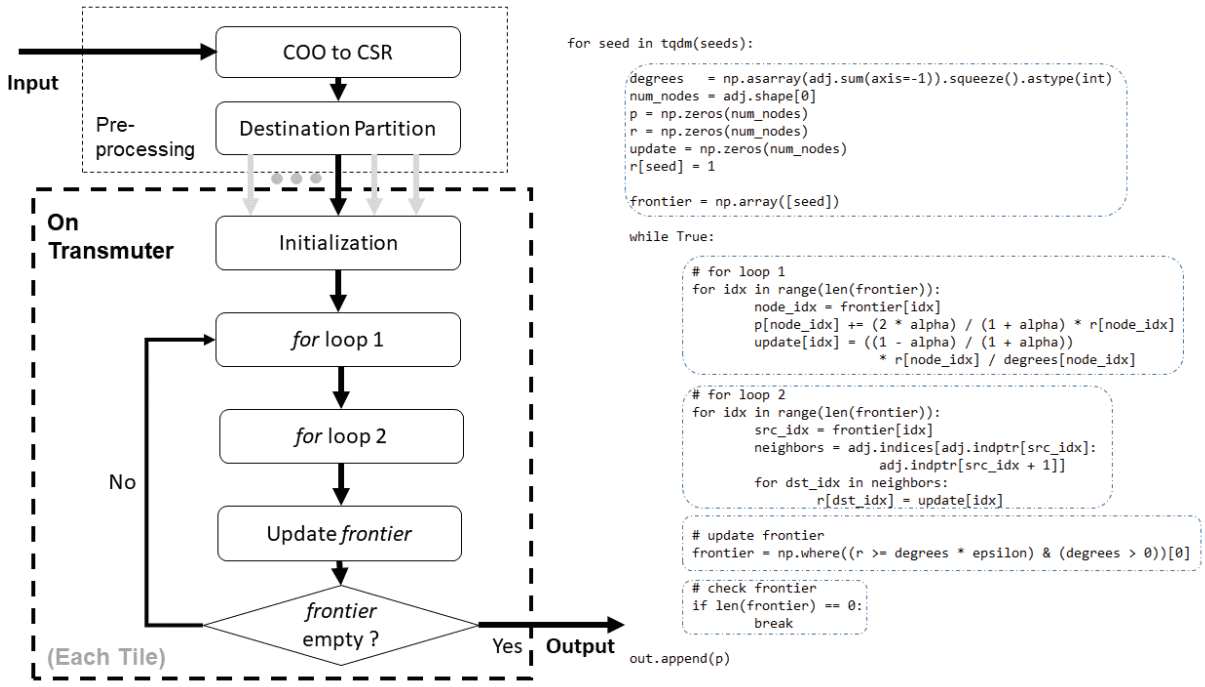


Figure 103: Total Runtime, Energy and EDP Comparison for Density = (a) 0.0034, (b) 0.034 and (c) 0.34 Cases

5.5 LGC pr_nibble Analysis – Phase 1

PageRank-Nibble (PR-N) is a variant of PageRank (PR) where PageRank is performed to a selected subset of vertices. Instead of computing the global importance scores, PR-N computes the importance score of a vertex from a specific seed vertex. The final local cluster decision is based on the score of each node.

In this algorithm, two vectors are used to record the properties of each node: *p* vector is used to record the importance scores and *r* vector is used to store the residues. In each iteration, the algorithm first determines the active vertices based on *r* vector. For the active vertices, the algorithm adds the scaled *r* values to the corresponding importance scores in *p* array, and then updates the *r* values of their destination vertices.



(a) (b)

Figure 104: (a) Flow Diagram of PageRank-Nibble, and (b) the Revised Algorithm

There are two *for* loops in each iteration: *for* loop 1 updates *p* array for the active vertices, and *for* loop 2 updates *r* values for their children vertices. In the original Python code, the *update* variable computed in *for* loop 2 only depends on the old *r* values of the active vertices and is repeated for all destination vertices. To avoid the repeated computations, we move this operation to *for* loop 1. We initialize an *update* vector with the same size as *frontier* array, perform the update computation for each active vertex, and then write the values into the *update* vector. In the *for* loop 2, we directly read values from the *update* vector. Since the old *r* vector is no longer needed in the *for* loop 2, we can remove the *r_prime* vector. Figure 104 shows the flow diagram and the revised algorithm.

5.5.1 Mapping on Transmuter

The default configuration of Transmuter is 4 tiles with 16 GPEs in each tile. LGC is repeatedly called to find local clustering starting from different seed vertices. Since each run is independent, we parallelize the computation by using each tile to run PR-N from a specific seed vertex.

With this mapping, we initialize one copy of *p* vector and *r* vector in each tile. We configure L-1 memory as shared cache and L-2 memory as private cache. Since there is no data dependency between GPEs in different tiles, this mapping does not require cross-tile synchronization or cache flush.

In *for* loop 1, there is neither data dependency nor over-writing risk. We evenly distribute the workloads, i.e. the vertices in the *frontier* vector, to different GPEs. In *for* loop 2, we store the graph information using the compressed sparse row (CSR) format (same as the provided Python

code). The algorithm goes from the source vertex to the destination vertices through *ind_ptr* and *indices* vectors. We divide the workloads through a destination-based partition to avoid over-writing.

The input format of the graph provided by DARPA is coordinate format (COO). Prior to the initialization phase, we transform the input data into compressed sparse row (CSR) format and perform destination-based partitioning. The partitioned graphs are then fed to the GPEs by the LCP.

5.5.2 Destination-based Partitioning

We apply balanced destination-based partitioning to balance the workloads of GPEs within each tile [18]. This scheme first arranges all edges according to their destination vertices, evenly divides the edges to different groups based on the number of GPEs per tile, and then sends the range of corresponding destination vertices in each group to each GPE. The partition scheme ensures that each GPE processes same number of edges. In *for* loop 2, the GPEs traverse all the active source vertices. If the destination vertex is within its partition, it updates the *residue* values, otherwise, it continues to the next destination. This partition scheme avoids data over-writing and the use of locks.

The destination-based partitioning scheme is performed in the pre-processing phase and is not implemented on Transmuter.

5.5.3 Pre-fetcher for Graph Data Structure

Since the accesses to *ind_ptr* and *indices* vectors are random memory accesses, it results in large access latency. To reduce cache misses, we apply a pre-fetcher that is customized for graph data structure [19]. After the access to *frontier* vector is sensed on the bus, the pre-fetcher looks for the next entry in the *frontier* vector that will be accessed in the near future. Based on this 'future' entry, the pre-fetcher fetches the cache-lines of the corresponding *ind_ptr* vector, *indices* vector, and *r* vector. After the GPE comes to this 'future' entry, all its needed data are in the cache, thus the cache miss is avoided.

5.5.4 Performance Results on DARPA Data Set

Before the algorithm starts on Transmuter, we pre-process the input data using *Python* programming on the host machine. The operations in the pre-processing include: (1) transform the graph from COO to CSR format; (2) perform the destination-based partition. With an Intel Core i7-6770HQ machine with clock frequency of 2.6 GHz and RAM size of 32 GB, the execution time of pre-processing operations is 0.241 s.

On Transmuter, we use one LCP to read partition results and the data into the DRAM. The execution time of this process together with all vector initialization operations is 3.8 ms, which is around 2% of the overall execution time on Transmuter. The energy consumption of LCP in this process is 7.3×10^{-4} J.

We run PR-N on Transmuter using the data set provided by DARPA. The simulation results are shown in Table 26. The power number is estimated based on 14 nm technology node. The

execution time is for 50 seed vertices. Compared to the execution without pre-fetcher, the pre-fetcher increases the GOPS/W by 6.7%.

Table 26: Performance Data for PageRank-Nibble for 50 Seeds

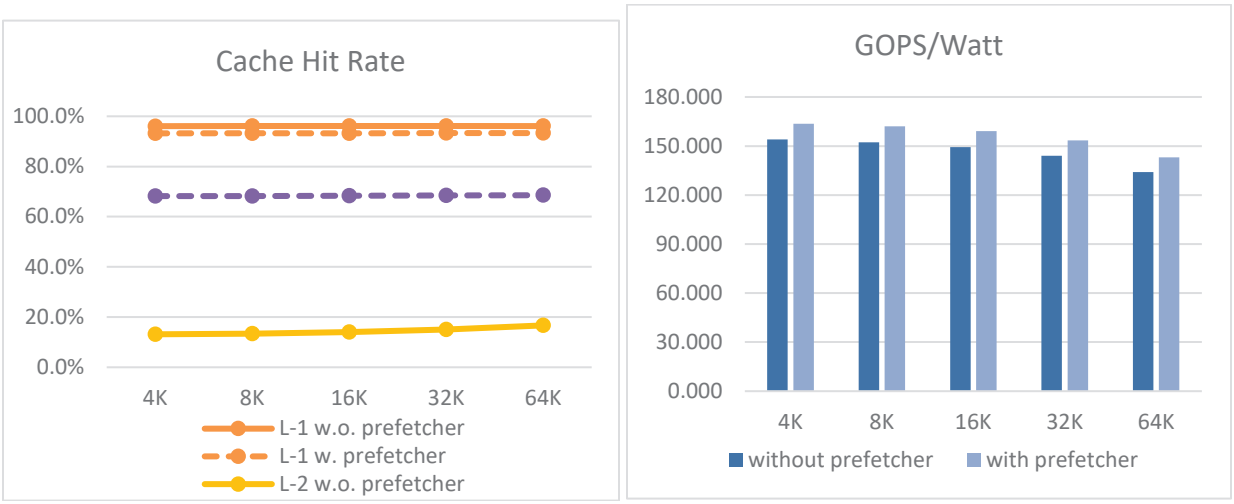
	GOPS /Watt	GFLOPS /Watt	Execution Time (s)	Total Energy (J)	Sim. Cycles	Cache Hit Rates (%)		%System Simulated
						L-1	L-2	
Without Prefetcher	139.63	2.71	0.24+0.224*	0.0532	2.2×10^8	96.18	16.37	100%
With Prefetcher	148.51	3.07	0.24+0.192*	0.0469	1.9×10^8	93.51	68.42	100%

* 0.24 is the time taken by Python pre-processing executed on the host machine.

The rest of execution time reported in this document only consider the execution time on Transmuter.

5.5.5 Effect of L-2 Cache Bank Size on Performance

We run simulations for 20 seed vertices with L-2 bank size of 64 KB, 32 KB, 16 KB, 8 KB, and 4 KB. The execution time does not change for different cases. The cache hit rates and GOPS/Watt are shown in Figure 105. As the size of L-2 memory bank reduces, the cache hit rates remain the same, but GOPS/W increases slightly because of lower memory power consumption. So the execution efficiency of PR-N does not change much with different L-2 bank size.



(a) (b)

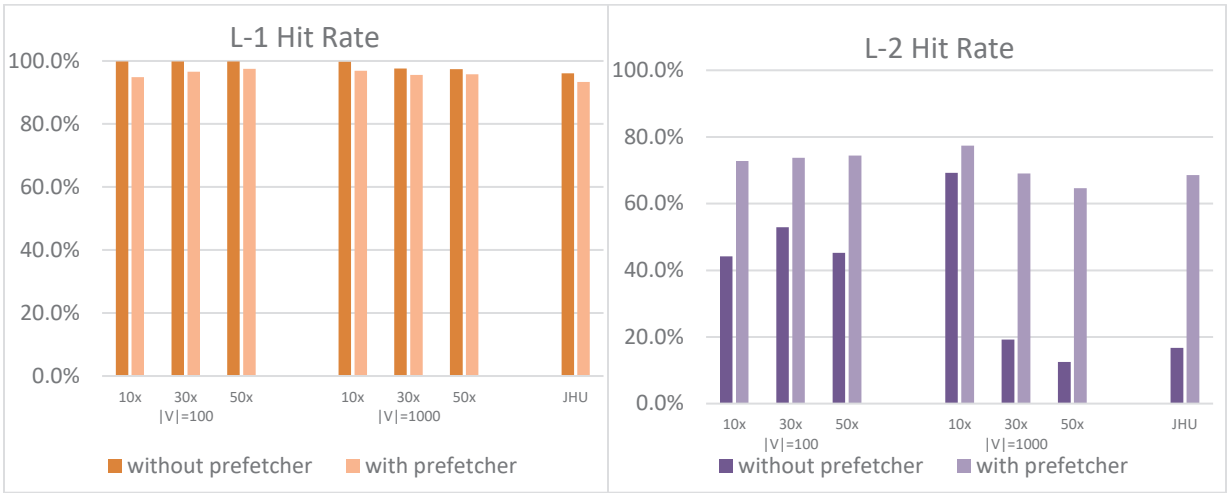
Figure 105: (a) Cache Hit Rates and (b) GOPS/W of PR-N for different L-2 Bank Sizes
GOPS/W does not change significantly with L-2 bank size. For large L-2 size, the hit rate without the pre-fetcher increases slightly.

5.5.6 Performance for Different Input Data Sizes

We investigate the GOPS/W for different graph sizes. The input graphs are randomly generated using uniform distribution. Table 27 shows the input graph sizes. 'JHU' is the graph provided by DARPA. The number of seed vertices is 20. Figure 106 shows the cache hit rates. The prefetcher increases the cache hit rates significantly for graphs with more than 25000 edges. Figure 107 shows the execution time and GOPS/Watt. For the same number of vertices, GOPS/W and the execution time increase as the number of edges increases.

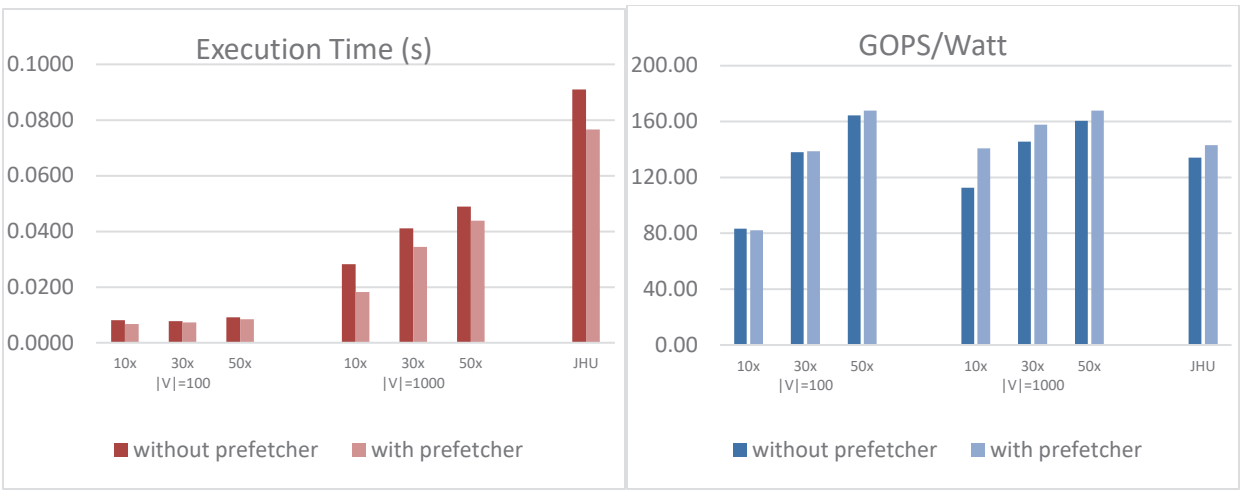
Table 27: Number of Vertices and Edges in the Simulated Input Graphs

|V|	100			1000			5158
|E|	1000	3000	5000	10000	30000	50000	373144
|E|/|V|	10	30	50	10	30	50	JHU



(a) (b)

Figure 106: (a) L-1 and (b) L-2 Hit Rates for PR-N Running with Different Graph Sizes
The pre-fetcher increases the L-2 cache hit rate significantly for large graphs.



(a) (b)

Figure 107: (a) Execution Time and (b) GOPS/Watt of PR-N Running for 20 Seed Vertices with Different Graph Sizes.

5.5.7 Extrapolation Performance on a Larger System

To project for larger configurations, we run the PR-N implementation with DARPA workloads for 24 seed vertices on the following three configurations:

(1) **2x8**: 2 tiles and 8 GPEs per tile;
(2) **4x16**: 4 tiles and 16 GPEs per tile;
(3) **8x16**: 8 tiles and 16 GPEs per tile.

We test the GOPS/Watt, GFLOPS/Watt, execution time, synchronization time, bandwidth, energy, and cache hit rates for each execution. Table 28 presents the performance generated using different configurations.

Table 28: Performance of PR-N for 24 Seeds on Different Configurations

Config.	GOPS/W	GFLOPS/W	Execution Time (s)	Synchronization Time (s)	Bandwidth Util.(%)	Total Energy (J)	Cache Hit (%)	
							L-1	L-2
2 x 8	121.45	4.36	0.1672	0.0376	0.82	0.015	**88.20**	90.76
4 x 16	143.08	2.98	0.093	0.0238	1.56	0.022	93.31	68.68
8 x 16	131.07	2.69	0.056	0.0164	2.76	0.025	92.84	68.96

The 8x16 configuration has the shortest execution time, as expected. The 2x8 configuration has the smallest energy consumption, due to the lower power consumption of memory banks. Figure 108 shows the trend for execution time, synchronization time, and bandwidth utilization based on the data executed using three configurations.

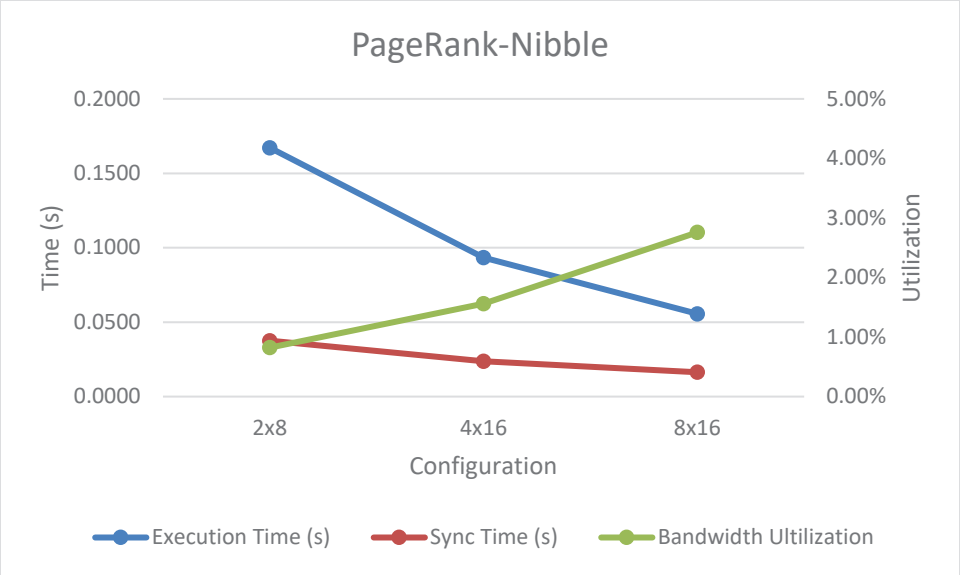


Figure 108: Trend of Execution Time, Synchronization Time, and Bandwidth Utilization of Page Rank Nibble

As the numbers of GPEs increase, the execution time reduces as expected. However the execution time does not scale proportional to the number of GPEs but scales with the number of tiles. In our implementation, each tile is assigned a seed and so more seeds can be processed at a time when there are more tiles. The bandwidth utilization and the ratio between synchronization time and the execution time also increase as more GPEs are used for computation. In PR-N, since the computation of the new frontier is sequential, we use only one GPE to perform this computation. On the other hand, although we apply the balanced destination-based partitioning scheme, the workload of each GPE still depends on the vertices in the frontier array. So, the synchronization time ratio of PR-N is higher than other workflows. From this trend, we can estimate that the ratio of synchronization time will increase as the number of GPEs increases.

5.5.8 Extrapolation Performance on a Larger Data Set

Since the executions of PR-N starting from different seed vertices are independent, the execution time increases linearly with the number of seeds if the number of seeds is a multiple of number of tiles. For example, the time for a 4x16 configuration to run 24 seeds and 50 seeds (same as 52 as there is 4 tiles) are 0.093 s and 0.192 s, respectively. Based on this execution time trend, we can estimate the time to run larger number of seeds will increase proportionately.

The time taken for each computation corresponding to a seed vertex is not the same. This is because the number of vertices in the frontier array and the number of iterations to find the local cluster depend on the graph characteristics. Basically, the number of iterations required to reach the stopping criteria depend on the graph connectivity. So the execution time is also a function of connectivity of the input graph.

5.6 LGC ISTA Analysis – Phase 1

Iterative shrinkage-thresholding algorithm (ISTA) for L1-regularized PageRank is an optimization algorithm to solve the personalized PageRank problem [20]. Compared to PR-N, it directly uses the adjacency matrix to perform computations instead of using the graph traTransmuter v1.0l pattern.

We divide the algorithm into two parts: the initialization part and the iteration part. The initialization part consists of computing square root and reciprocal square root of degree vectors, normalizing the adjacency matrix, and computing iteration coefficients. Here, the most computation-intensive part is normalizing the adjacency matrix, which consists of two matrix-matrix multiplications. The iteration part consists of two vector-additions, a matrix-vector multiplication, and several scalar operations in each iteration. Figure 109 gives the high-level description of the algorithm.

The two matrix-matrix multiplications in the initialization part include multiplying the *Dn_sqrt* matrix, which is the reciprocal of degrees's square root, on both left and right side of the adjacency matrix *Q*. Since *Dn_sqrt* only has non-zero elements on the diagonal, we can use its vector form, *dn_sqrt*, in the implementation. Then these two matrix-matrix multiplication operations can be simplified into:

$$Q[i][j] = dn\_sqrt[i] * dn\_sqrt[j] * Q[i][j] \quad (1)$$

Compared to performing dense matrix-matrix multiplications, this implementation saves much memory access overhead.

5.6.1 Mapping on Transmuter

Since the execution starts from different seed vertices are independent, we parallelize ISTA by using each tile to run with a specific seed vertex. With the default configuration of 4 tiles and 16 GPEs in each tile, we use each tile to run the algorithm starting from a seed vertex. The L-1 memory in all tiles are configured as shared cache and the L-2 memory is configured as private cache.

The initialization part of ISTA has large number of floating-point operations. In this part, the only operation that involves the starting seed vertex is computing the initial gradient vector, *grad0*, which has only one non-zero element. So, to avoid repeated computations, we move the computation of *grad0* to the iteration part. In this way, we only run the initialization part once and share the data to all executions of different seed vertices. We use all 4 tiles to compute the initialization part. Figure 109 shows the flow diagram and the revised algorithm.

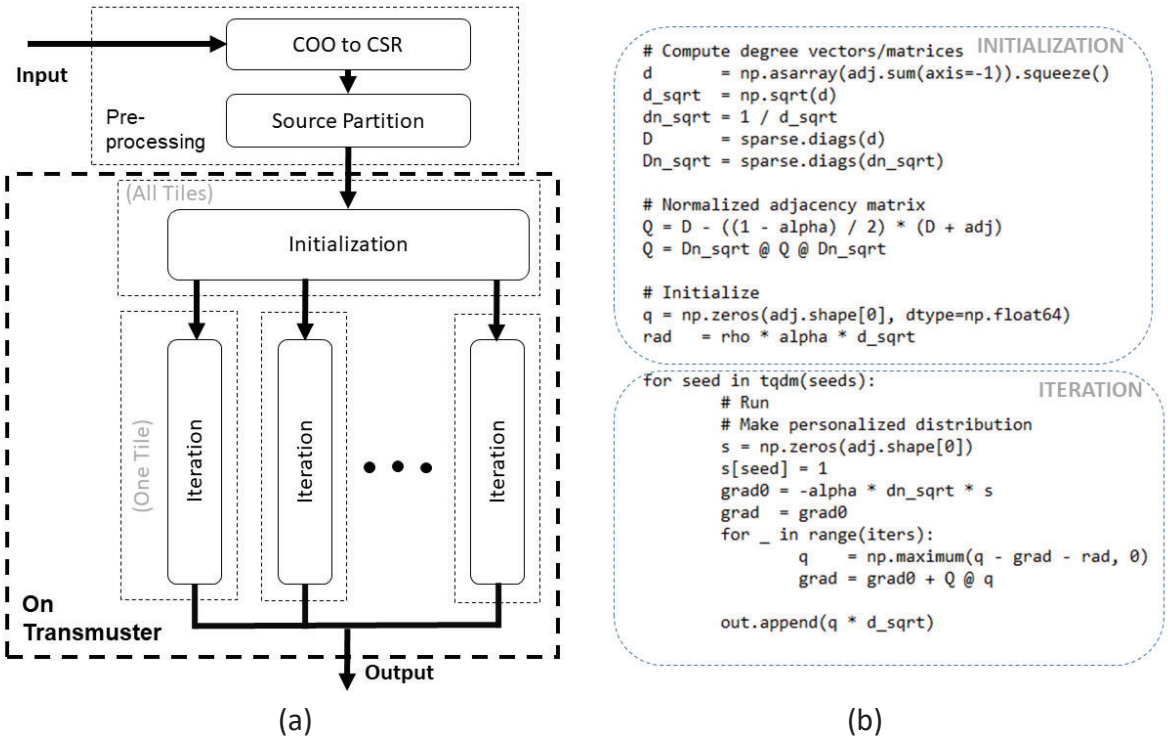


Figure 109: (a) Flow Diagram and (b) the Revised Algorithm of ISTA

We store the normalized adjacency matrix Q in the compressed row (CR) form [1], where a list of arrays is used to represent a matrix. Each array stores the indices and values of nonzero elements for a row of Q. We store *dn_sqrt*, *d_sqrt*, *grad, rad* and *q* vectors in dense form as most of their elements are non-zero.

In the initialization part, we implement the matrix-matrix multiplication that normalizes the adjacency matrix as in Equation (1). In the iteration part, we implement the matrix-vector multiplication using the sparse-matrix-dense-vector multiplication. Since the number of non-zero elements in each row is same as the degree of the corresponding node, we apply the source-partition scheme to distribute the rows of Q matrix. Hence the number of multiplications run by each GPE is balanced.

5.6.2 Performance Results on DARPA Data Set

Before the algorithm starts on Transmuter, we pre-process the input data using *Python* programming on the host machine. The operations in the pre-processing include: (1) transform the graph from COO to CSR format; (2) perform the source-based partition algorithm. With an Intel Core i7-6770HQ machine with clock frequency of 2.6 GHz and RAM size of 32 GB, the execution time of pre-processing operations is 0.246 s.

On Transmuter, we use one LCP to read partition results and the data into the DRAM. The execution time of this process together with all vector initialization operations is 6.1 ms, which is

around 3.3% of the overall execution time for 20 seed vertices on Transmuter. The energy consumption of LCP in this process is $1.17x10^{-3}$ J.

We run ISTA algorithm on Transmuter using the provided data set. The simulation results are shown in Table 29. The execution time is for 20 seeds. The time of algorithm initialization part is 0.5% of the total execution time whereas most of execution time is taken by iteration part. As the execution for different seed vertices are independent, the execution time for 500 seeds is likely to be 25 times the execution time of 20 seeds. Considering that most operations in ISTA are floating-point operations, we count the number of floating-point operations.

Table 29: Performance of ISTA for 20 Seeds

	GOPS /Watt	GFLOPS /Watt	Execution Time (s)	Total Energy(J)	Simulated Cycles	Cache Hit Rates (%)		Percentage of System Simulated
						L-1	L-2	
ISTA	106.08	46.64	0.25+0.191*	0.0415	$1.91x10^{8}$	91.86	6.32	4%

*0.25 is the time taken by Python pre-processing.

In the rest of this section, we only report the execution time of Transmuter.

5.6.3 Effect of Different L-2 Cache Bank Sizes on Performance

We run the algorithm with different L-2 memory bank sizes. Figure 110 shows the cache hit rates and GFLOPS/W as a function of different L-2 bank sizes. As L-2 bank sizes increases from 4K to 64K, the GFLOPS/W reduces slightly, which is due to the higher power consumption contributed by larger L-2 memory.

(a) (b)

Figure 110: (a) Cache Hit Rates, and (b) GFLOPS/W as a Function of L-2 Cache Bank Sizes

GFLOPS/Watt reduces for large L-2 bank size.

5.6.4 Performance for Different Input Data Sizes

We investigated the performance for different graph sizes. Figure 111 shows the L-1 and L-2 cache hit rates for different graph sizes. The L-2 hit rate reduces when the graph size is large. Figure 112 shows the execution time and GFLOPS/Watt. Since the edge number equals the number of non-zero elements in the Q matrix, the execution time is proportional to the number of edges. Thus, GFLOPS/W increases with the number of edges in the graph.

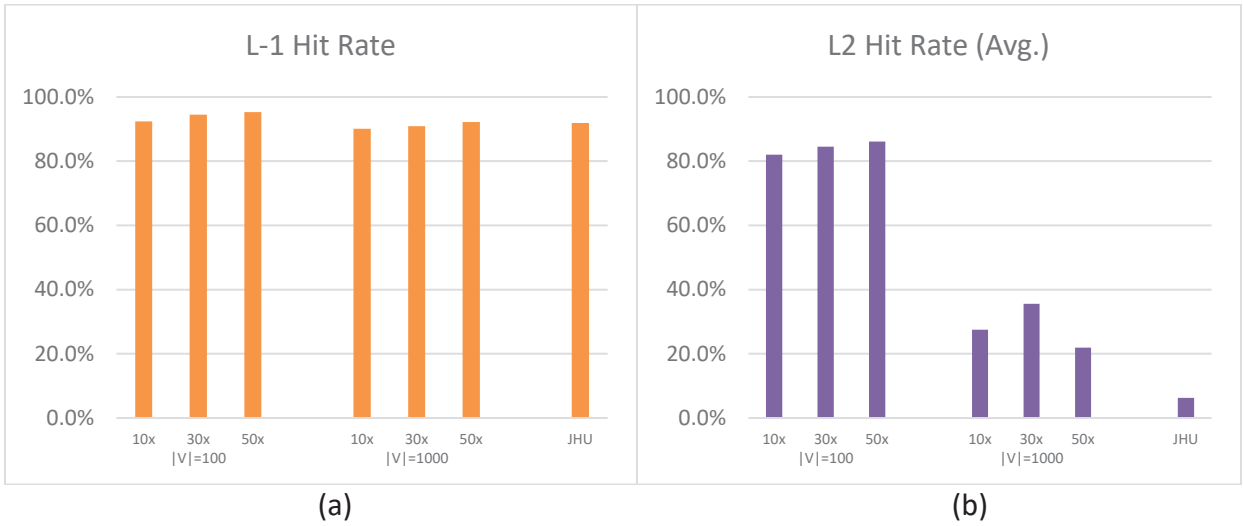


Figure 111: (a) L-1 Hit Rate and (b) L-2 Hit Rate as a Function of Different Input Graphs Sizes

The L-2 hit rate is low for large graphs with more than 10000 edges.

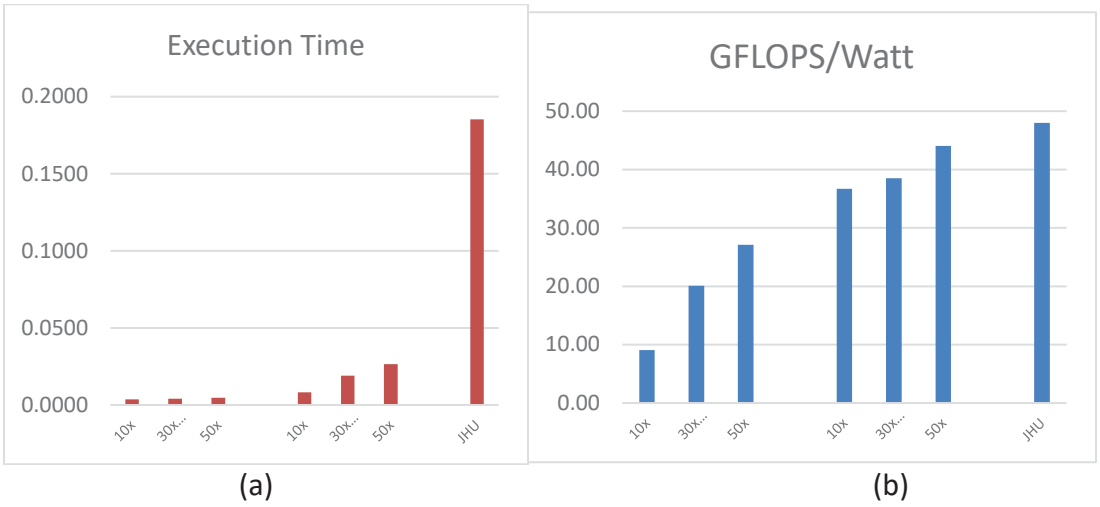


Figure 112: (a) Execution Time and (b) GFLOPS/Watt as a Function of Different Input Graph Sizes

As the number of edges increases, both execution time and GFLOPS/Watt increase.

5.6.5 Extrapolation Performance on a Larger System

To project for larger configurations, we run the ISTA implementation using DARPA workloads for 24 seed vertices on the following three configurations:

(1) **2x8**: 2 tiles and 8 GPEs per tile;
(2) **4x16**: 4 tiles and 16 GPEs per tile;
(3) **8x16**: 8 tiles and 16 GPEs per tile.

We test the GOPS/Watt, GFLOPS/Watt, execution time, synchronization time, bandwidth, energy, and cache hit rates for each execution. Table 30 presents the performance generated using different configurations.

Table 30: Performance of ISTA for 24 Seeds on Different Configurations

Config.	GOPS/W	GFLOPS/W	Execution Time (s)	Synchronization Time (s)	Bandwidth Util.(%)	Total Energy (J)	Cache Hit (%)	
							L-1	L-2
2 x 8	92.74	40.73	0.688	0.0239	3.47	0.057	76.85	71.37
4 x 16	108.36	47.78	0.224	0.0107	4.79	0.049	91.90	6.12
8 x 16	89.49	39.25	0.149	0.0093	6.20	0.059	91.79	7.57

In the three configurations, configuration 8x16 has the shortest computation time as expected. Configuration 4x16 has the highest GOPS/Watt and GFLOPS/Watt and also has the lowest energy consumption.

Figure 113 shows the trend for execution time, synchronization time, and bandwidth utilization based on the data executed using three configurations.

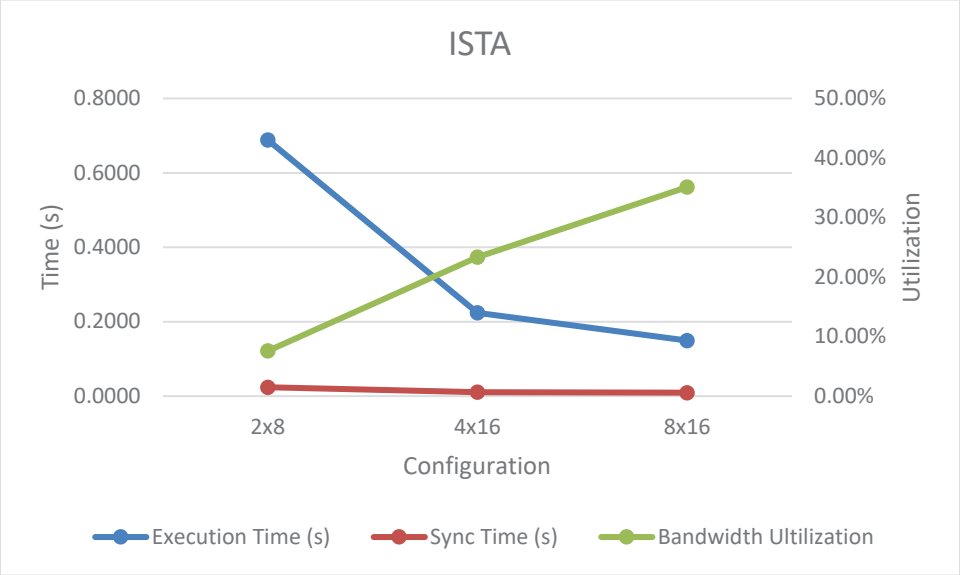


Figure 113: Trend of Execution Time, Synchronization Time, and Bandwidth Utilization of ISTA

The execution time reduces with increase in the number of GPEs almost linearly. This implies that the algorithm has been parallelized well. The ratio between the synchronization and total execution time is significantly lower than PR-N. It too increases as the number of GPEs increases.

This is because the workload distribution is not even when the number of GPEs is large and some GPEs have to wait for others in an iteration. The energy consumption is lowest for the (4x16) configuration. Thus if the default configuration is 8x16, for energy-constrained applications, it would be best to hibernate four of the tiles.

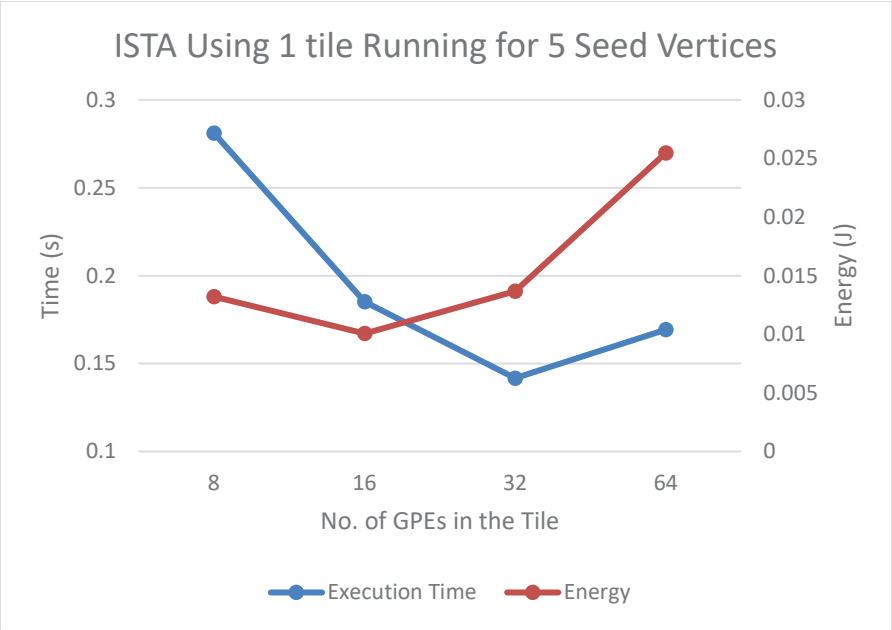


Figure 114: Trend of Execution Time and Energy Consumption for 5 Seeds of ISTA using DARPA Workloads as a Function of Different Number of GPEs in the Tile
The number of tiles is 1.

Figure 114 shows the execution time of ISTA for 5 seeds using only one tile but different number of GPEs in the tile. The execution time reduces as the number of GPEs increase from 8 to 32 and then increases as the number of GPEs is larger than 32. The energy consumption reduces from 8 GPEs to 16 GPEs, and then starts to increase for larger number of GPEs per tile. For a configuration with more than 32 GPEs in each tile, we can use the reconfiguration to shut off the excessive GPEs to achieve high computation efficiency.

5.6.6 Extrapolation Performance on a Larger Data Size

Compared to the time taken by iteration part, the algorithm iteration part is negligible. Since the execution of each seed vertex is independent and the number of iterations is fixed, the execution time of ISTA is proportional to the number of seed vertices. This trend can be verified through the execution time of 20 seed vertices and 24 seed vertices presented in Table 1 and Table 2. Based on this trend, we can estimate the execution time for 500 seed vertices to be around 4.78 s.

On the other hand, the number of operations in the matrix-vector multiplication depends on the number of non-zero elements in the adjacency matrix, which is the number of edges in a graph. With more edges, the number of operations in each iteration increases, and the execution time becomes larger. Therefore, the execution time also linearly increases with the number of edges in the graph.

The execution time increases linearly with the number of seed vertices and the number of edges in the graph.

5.7 Convnet Analysis – Phase 1

ConvNet is a simplified Residual Convolutional Neural Network (ResNet) [21]. This ResNet consists of one convolutional layer with batch normalization and ReLU for feature extraction, three residual blocks with max pooling and a linear (fully connected) layer. In each residual block, the main path is a convolution layer with batch normalization and ReLU, and the bypass path is convolution layer using 1x1 kernel to change the size of the input. The output of the residual block is the summation of values from the main path and the bypass path.

In ConvNet, key kernels are 2D convolution using kernel size of 3x3 and 1x1, batch normalization, max pooling and fully connected layer. Next, we present the detailed implementations of these kernels.

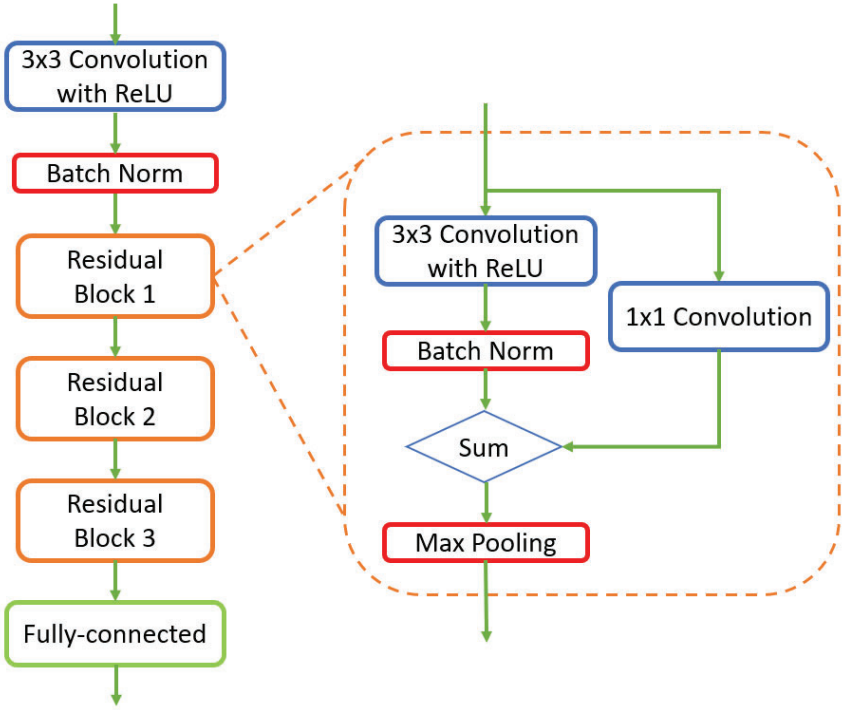


Figure 115: ConvNet Software Architecture

5.7.1 Key Kernel Implementations on Transmuter

3x3 2D Convolution (Forward): 3x3 2D convolution is implemented in a row stationary style using 2D systolic array mode. Here L1 cache is configured to support systolic array mode and L2 cache is in shared cache mode. 4 GPEs are grouped into one computation group where 3 GPEs compute 2D convolution using systolic array and 1 GPE computes ReLU function and the local mean. In this implementation, 2D convolution is computed row by row. The 3x3 kernel is distributed to 3 GPEs. Each GPE stores a row of weights and the input rows flows between GPEs.

In each round of computation, each GPE carries out 1D convolution between one input row and the kernel row to get a partial sum. This partial sum is sent to the next GPE which updates it and sends the updated partial sum to the next GPE.

1x1 2D Convolution (Forward): 1x1 2D convolution computation uses L1 and L2 cache as shared cache. Each input channel is assigned to one GPE and the GPE multiplies the values in the input channel with the weight.

Backward Propagation of 3x3 2D Convolution: The backward propagation of 3x3 2D convolution is computed in two steps. In both steps, the L1 and L2 are configured in shared cache mode. The first step creates a set of active entry matrices using the output gradient matrix. In this step, each GPE receives one weight and generates the active entry matrix. The active entry matrix indicates the area where a kernel weight is used during the 2D convolution. With active entry matrices, the gradient of kernel and input can be computed. First, each GPE is assigned active entry matrix of one weight and carries out element-wise multiplication between active entry matrix and input matrix to get product matrix. Then each GPE adds the elements in the product matrix to get the weight gradients. To compute the gradient of input, each GPE multiplies the active entry matrix with the kernel weights, then carries out element-wise summation with the product matrix.

Batch normalization: Batch normalization is computed after the 3x3 2D convolution. This step is done in the shared cache mode and so reconfiguration has to be done before proceeding to this step. First, LCP collects all local sums and computes the global mean and variance of the batch. The mean and variance are stored in addresses where all GPEs have accesses to. Then the outputs are distributed to GPEs channel by channel and each GPE computes the batch-normed output of the channel. The backward pass of batch normalization also uses L1 and L2 in shared cache mode. Channels are distributed to each GPE along with mean and variance of the batch. Each GPE then computes the gradient of one channel in the output.

Max Pooling: For both forward and backward pass of max pooling, L1 and L2 are configured in shared cache mode. The computation of max pooling is parallelized by distributing channels of the output to each GPE. Each GPE receives one channel of the output. In forward pass, GPE computes the max pooling and stores the indices of every local maximum in a matrix which has the same size as the output. The index matrices are used to compute the backward pass of max pooling. In backward pass, each GPE is allocated one channel of the output gradient. According to the indices in the index matrix, the GPE writes the gradient into the output gradient matrix.

5.7.2 Performance Results on DARPA Data Set

We run the algorithm on Transmuter using the provided CIFAR10 data set. The bath size we use is 16, while the batch size specified by DARPA is 128.

Initialization
The initialization for ConvNet is done by LCP. First, it does the address mapping for all the data including input, output, weight, gradient, etc. After the addresses are allocated, LCP sends out the addresses to all the GPEs. Then the LCP loads the input image to the DRAM. The initialization is done at the beginning of every batch.

Execution Time for batch size of 16:

1. **Kernel View**

Table 31: Runtime Breakdown for each Kernel on Batch Size 16 (tabular)

Kernel	Execution time/s	%
Initialization	0.00324	6.33
3x3 Conv2d Forward	0.010699	20.92
1x1 Conv2D Forward	0.000213	0.42
Max Pooling Forward	0.000862	1.69
Batch-Norm Forward	0.002043	3.99
1x1 Conv2D Backward	0.000499	0.98
3x3 Conv2D Backward	0.029998	58.65
Max Pooling Backward	0.000681	1.33
Batch-Norm Backward	0.002837	5.55
Padding and FC layer	0.0000732	0.14
Total Time	0.0511452	100

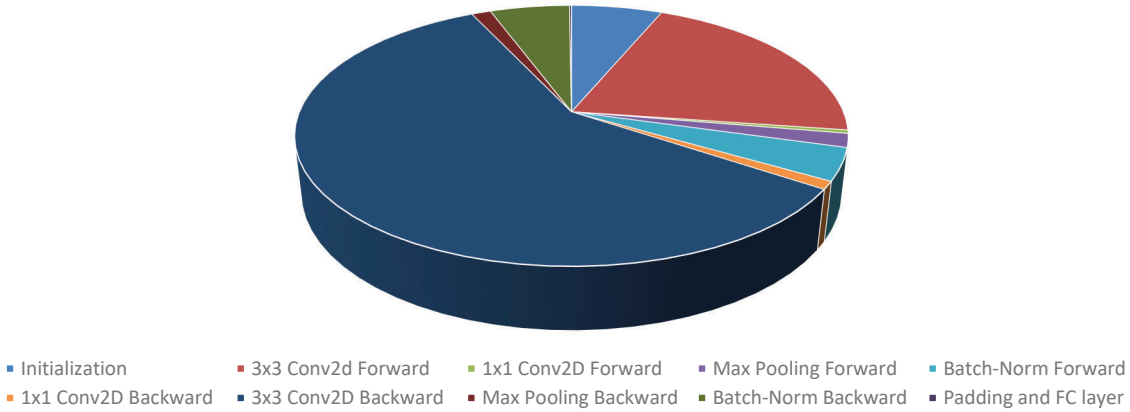


Figure 116: Runtime Breakdown for each Kernel on Batch Size 16 Pie Chart

2. **Layer View**

Table 32: Runtime Breakdown for each Layer on Batch Size 16 Tabular

Layer	Execution time/s	%
Initialization	0.00324	6.33
3x3 convolution	0.009716	18.99
residual block1	0.015205	29.73
residual block2	0.012267	23.98
residual block3	0.010712	20.94
Fully connected	0.0000052	0.01
Total Time	0.0511452	100

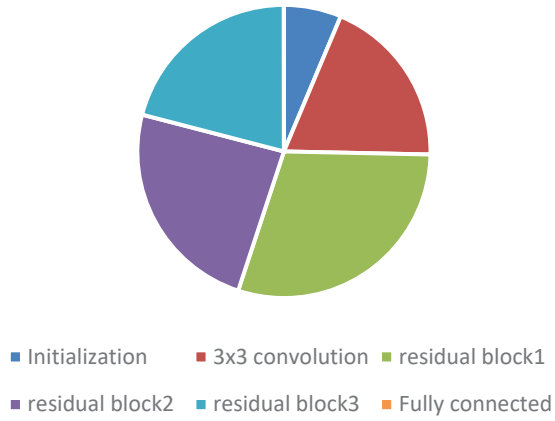


Figure 117: Runtime Breakdown for each Layer on Batch Size 16 Pie Chart

Table 33: GOPS and GOPS/W and L1/L2 Miss Rate

Kernel	Execution time/ms	En-ergy/mJ	GFLOPS	GOPS	GOPS/W	L1 Hit Rate (%)	L2 Hit Rate (%)
Initialization	3.24	0.65	0.01	1.53	7.58	-	55.57
3x3 Conv2D Forward	10.699	1.92	0.55	2.51	13.96	-	72.67
1x1 Conv2D Forward	0.213	0.04	2.22	5.79	30.83	86.74	5.81
Max Pooling Forward	0.862	0.16	1.50	14.19	76.08	83.69	26.16
Batch-Norm Forward	2.043	0.45	0.79	45.36	208.03	96.03	3.28
1x1 Conv2D Backward	0.499	0.10	6.31	14.00	68.85	96.49	22.90
3x3 Conv2D Backward	29.998	5.48	1.46	5.67	31.06	94.17	19.89
Max Pooling Backward	0.681	0.12	1.81	3.11	17.79	84.38	27.31
Batch-Norm Backward	2.837	0.51	0.61	3.72	20.59	90.08	46.86
Overall	51.077	9.43	1.18	6.42	34.75	94.04	42.88

Table 34: SDH Performer Performance Table for Batch Size of 16

Workflow	GOPS/W	Execution time (s)	Simu-lated/ es-timated cycles	Total energy (J)	L1/L2 Cache hit rates (%)	Percentage of system simu-lated
ConvNet	34.75	0.051	51000000	0.00943	94.04 / 42.88	12.5%

5.7.3 Extrapolation Performance on Larger System

We run the implementations using three configurations and estimate the potential architectural enhancements. The input workload is the image dataset **provided by DARPA**, which has size of 32x32x3. For each configuration, we run **batch size of 16**.

The three configurations are:

(1) **1x16**: 1 tiles and 16 GPEs per tile;
(2) **2x16**: 2 tiles and 16 GPEs per tile;
(3) **4x16**: 4 tiles and 16 GPEs per tile.

We test the execution time, synchronization time, bandwidth, and the energy for each execution. Table 35 and Table 36 presents the performance generated using different configurations.

Table 35: Convnet Performance using Different Configuration

Config	Execution time (ms)	Energy (mJ)		GFLOPS	GOPS	GOPS/W	L1 Hit Rate (%)	L2 Hit Rate (%)
4x16	47.91	8.79		1.11	6.75	36.77	94.04	41.45
2x16	65.22	7.11		0.98	5.04	46.21	92.64	42.78
1x16	91.12	6.03		0.84	4.09	61.82	92.55	50.42

Table 36: Synchronization Performance for ConvNet

Con-fig.	Execution Time (ms)	Synchronization Time (ms)	Bandwidth Util. (%)	Energy (mJ)
1x16	91.12	0.054	3.18	6.03
2x16	65.22	0.074	3.41	7.11
4x16	47.91	0.175	4.05	8.79

Figure 118 shows the trend for execution time, synchronization time, and bandwidth utilization based on the data executed using three configurations.

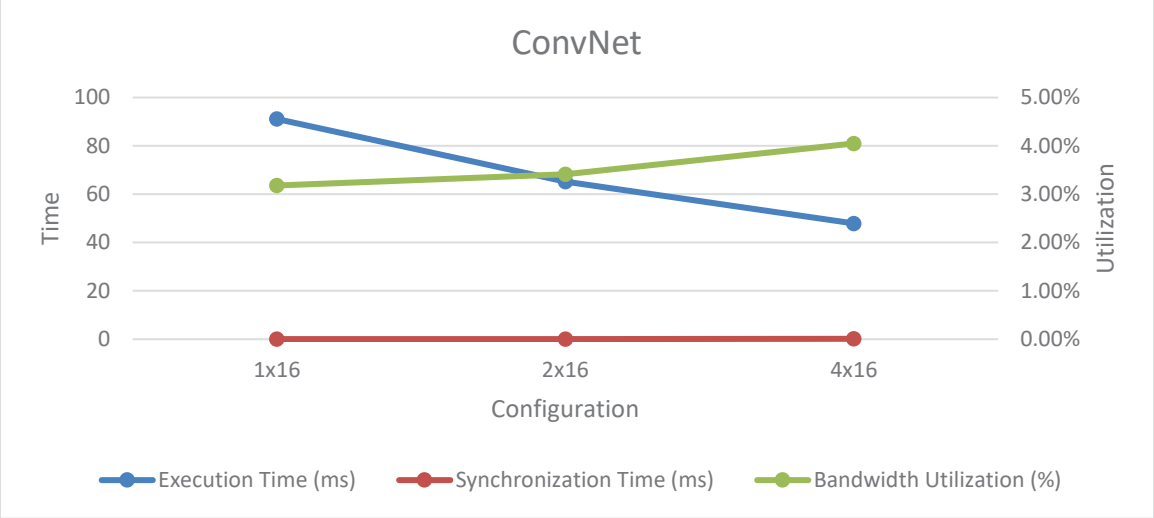


Figure 118: Trend of Execution Time, Synchronization Time, and Bandwidth Utilization of ConvNet

5.7.4 Extrapolation Performance on a Larger Data Size

In this simulation, input image size is 32x32x3. In most of the implementation, the GPEs are assigned with one channel of the input. In this case, if the number of input channels increase, the implementation will stay unchanged, but the workload will increase as the channel number increases. If the size of the input image increase, GPEs will require more L1 memory to process. If L1 cache size is not enough it will require more cache accesses which leads to lower cache hit rate, higher power consumption and higher execution time.

5.8 Phase 1 = Summary Table of Workflows Including Host Time/Power

Table 37: Overall TA-1 Evaluation Summary from Phase 1, Including Host Execution

	GOPS/W	Execution Time (s)	Total Energy (J)	Simulated/ estimated cycles	Cache L1/L2 hit rates	Percentage system simulated
Theoretical architectural peak	263.70					
graphsage	72.69	0.98	0.160	978325169	93.20 / 1.56	4%
ipnsw	42.40	1.58	0.299	1586244804	92.38 / 19.73	100%
recsys	169.12	1.06	0.245	1062000000	98.45 / 22.05	0.37%
sinkhorn_wmd	34.80	0.44	0.077	436100000	89.63 / 29.50	100%
LGC/pr_nibble	148.51	0.43	0.047	190000000	93.51 / 68.42	100%
LGC/ISTA	106.08	0.44	0.042	191000000	91.86 / 6.32	4%
convnet	34.75	0.05	0.009	51000000	94.04 / 42.88	12.5%

6 REFERNCES

[ ] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, “Outerspace: An outer product based sparse matrix multiplication accelerator”, in IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 724-736, 2018.
[ ] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, “Rigel: an architecture and scalable programming interface for a 1000-core accelerator”. In ACM SIGARCH Computer Architecture News, pp. 140–151, 2009,ACM.
[ ] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. “SODA: A Low-Power Architecture for Software Radio”, In the 33rd Annual International Symposium on Computer Architecture (ISCA), pp. 89-101, 2009.
[ ] Wang, Hao, Weifeng Liu, Kaixi Hou, and Wu-chun Feng, "Parallel transposition of sparse data structures," In Proc. of the 2016 International Conference on Supercomputing, pp. 1-13, 2016.

[ ] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, and others. “The gem5 simulator”, ACM SIGARCH Computer Architecture News, Vol. 9, Issue 2, pp 1-7. 2011.

[ ] Z. Wang, G. Tournavitis, B. Franke, and M.F.P. O’Boyle, “Integrating profile-driven parallelism detection and machine-learning-based mapping,” ACM Transactions on Architecture and Code Optimization (TACO), Volume 11, Issue 1, February 2014, p. 2
[ ] P. Ginsbach, T. Remmelg, M. Steuwer, B. Bodin, C. Dubach, and M.F.P. O’Boyle, “Automatic matching of legacy code to heterogeneous APIs: An idiomatic approach” to appear in 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2018
[ ] P. Ginsbach, and M.F.P. O'Boyle, “Discovery and exploitation of general reductions: a constraint based approach”, In Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO), February 2017, pp. 269-280
[ ] C. Dubach, T.M. Jones, and M.F.P. O'Boyle, “Microarchitectural Design Space Exploration Using an Architecture-Centric Approach”, 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), December 2007, pp. 262-271
[ ] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13), June 2013, pp 519-530
[ ] V. Kiriansky, Y. Zhang, and S. Amarasinghe, “Optimizing Indirect Memory References with milk”, In Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16), September 2016, pp. 299-312
[ ] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, “Plasticine: A Reconfigurable Architecture for Parallel Patterns”, 44th International Symposium on Computer Architecture (ISCA), June 2017, pp. 389-402
[ ] K.J. Brown, H. Lee, T. Rompf, A.K. Sujeeth, C. De Sa, C. Aberger, and K. Olukotun, “Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel

Patterns", International Symposium on Code Generation and Optimization (CGO), March 2016. pp. 194-205
[ ] A.K. Sujeeth, K.J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages", ACM Transactions on Embedded Computing Systems (TECS), Volume 13 Issue 4, July 2014
Article No. 134
[ ] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation", 2017 International Symposium on Code Generation and Optimization (CGO), February 2017. pp. 74-85
[ ] J. Cavazos, G. Fursin, F.V. Agakov, E.V. Bonilla, M.F. P. O'Boyle, and O. Temam, "Rapidly Selecting Good Compiler Optimizations using Performance Counters", International Symposium on Code Generation and Optimization (CGO), March 2007, pp. 185-197
[ ] Z. Wang, and M.F.P. O'Boyle, "Partitioning streaming parallelism for multi-cores: a machine learning based approach", Parallel Architectures and Compilation Techniques (PACT), September 2010, pp. 307-318
[ ] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "GraphGrind: addressing load imbalance of graph partitioning", in Proceedings of the International Conference on Supercomputing, pp. 16, 2017.
[ ] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge", in Proceedings of the 2016 International Conference on Supercomputing, pp. 39-49, ACM.
[ ] K. Fountoulakis, R. Kimon, S. Farbod, J. Shun, X. Cheng, and M. W. Mahoney, "Variational perspective on local graph clustering", in Mathematical Programming, vol. 174, no. 1-2, pp. 553-573, 2019.
[ ] K. He, X. Zhang, S. Ren and J. Sun. "Deep Residual Learning for Image Recognition", The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770-778.

LIST OF ABBREVIATIONS, ACRONYMS, AND SYMBOLS

ACRONYM	DESCRIPTION
CF	Collaborative Filtering
CNN	Convolutional Neural Network
CSC	Compressed Sparse Column
DIG	Data indirection Graph
DVFS	Dynamic Voltage Frequency Scaling
FPU	Floating-Point Unit
GCN	Graph Convolutional Networks
GPE	General-Purpose Processing Elements
LCP	Local Control Processor
LRG	Least-Recently Granted
MICRO	Microarchitecture
MSHR	Miss-Status Holding Registers
PFHR	Prefetch-Status Handling Register
R2R	Register-to-Register
SE	Syscall Emulation
XCU	Crosspoint Control Unit